Dr. Dobb's Journal

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
#107 SEPTEMBER 1985 $2.95 (3.95 CANADA)

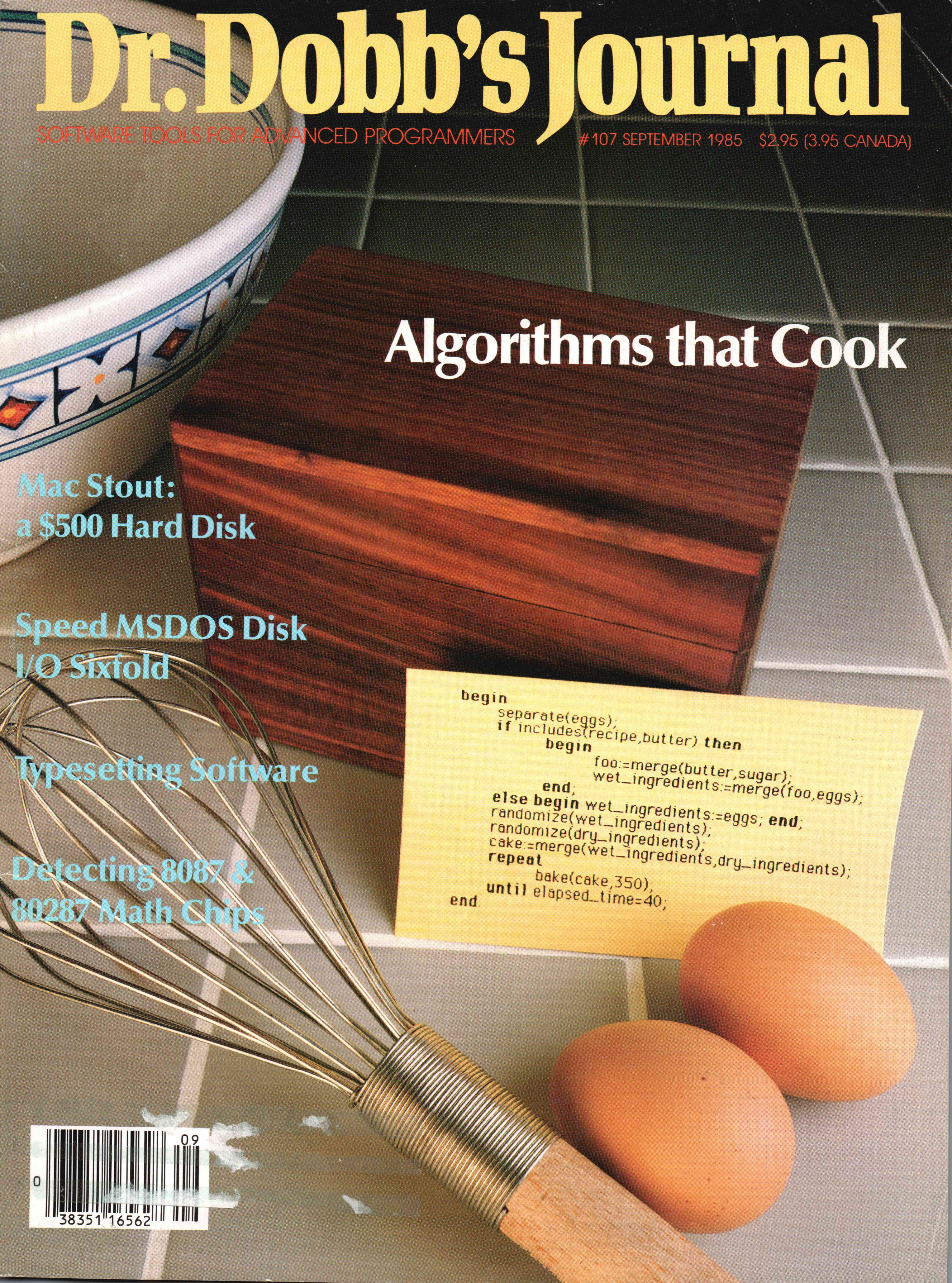
Algorithms that Cook

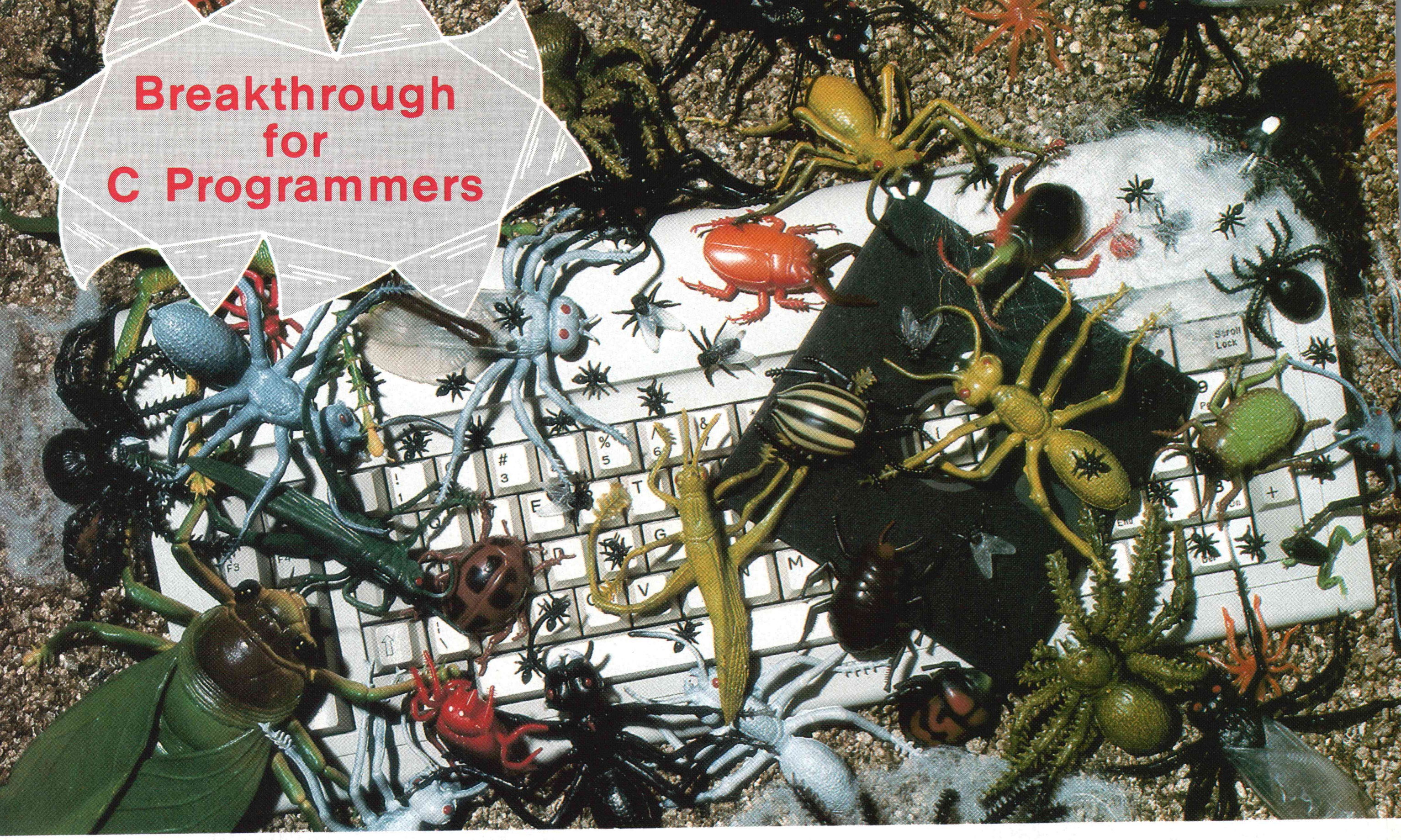
Mac Stout: a $500 Hard Disk

Speed MSDOS Disk I/O Sixfold

Typesetting Software

Detecting 8087 & 80287 Math Chips





H.E.L.P. Eliminates Every Bug known to Compilers . . . As well as a few other species

H.E.L.P. is a completely interactive C programming environment with three innovative full-sized features that will revolutionize the way you write code.

A Clean Compile — Guaranteed!

Say Good-bye to all compiler-type errors. **H.E.L.P.** 's built-in program checker (which would embarrass LINT) not only hunts down bugs . . . explains the proper syntax . . . gives examples of usage . . . but will even offer suggested corrections. If you want, **H.E.L.P.** will even make the corrections for you . . . at the touch of a key.

H.E.L.P. also finds semantic errors as well as poor style and inefficiencies. You can even check the portability of your code!

Multi-Window Editing

Open as many windows as you want . . . . there's no limitation . . . not even your own memory. Because **H.E.L.P.** uses a virtual memory system you can create programs larger than your machine capacity.

H.E.L.P.'s very powerful editor allows you the flexibility to work in several windows . . . with several files at the same time.

Save Keystrokes

Hundreds of commands are bound to the keyboard to give you fast execution.

Always be in Control

Not only can you develop code in many windows at the same time, but you can show (and refer to) important definitions in one window while creating in another. Or open a window and keep notes about your program . . . or type a memo . . . or a letter.

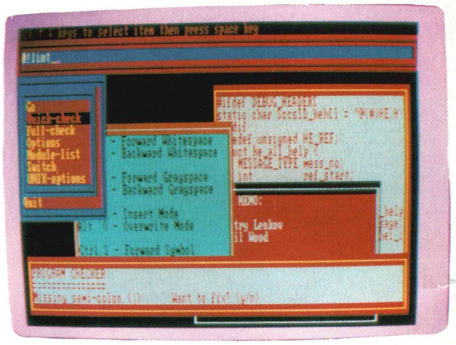
Increase your Productivity by 300% or More . . . . .

If you are a novice programmer, you'll begin writing code like an advanced programmer much faster with **H.E.L.P.**

Just imagine what **H.E.L.P.** will do for the ADVANCED PROGRAMMER.

You'll have more time to become creative with your algorithm (since **H.E.L.P.** will make sure your code compiles the "first time at bat").

H.E.L.P. tracks every step you make. If you are not sure about a command, just press a key, and you'll get the kind of help you need.



Check These Features

- *Multi-window environment*
- *Interactive program checking*
- *Check syntax, semantic, type usage, intermodule inconsistencies and portability*
- *Multi-file editing*
- *Intelligent help subsystem*
- *User-definable keyboard bindings*
- *Supports color and monochrome*
- ***H.E.L.P.** supports the full C Language*

NOW IN MS-DOS

Order now $395

EVEREST SOLUTIONS

Everest Solutions, Inc.
3350 Scott Boulevard
Building 58
Santa Clara, CA 95051
(408) 986-8977

living C-personal™

GIVES LIFE TO C PROGRAMMING

Living C™ is the fully integrated interactive programming environment for C. By replacing the headaches of programming in C with total control and understanding – whether new to C or an expert – Living C is the exciting solution to maximize your creativity and productivity.

$99

FULL SCREEN EDITOR

The Living C editor is a true full function commercial editor, fully menu-driven with help facilities on call. The editor is fully integrated and can be used at all times throughout the design, development, maintenance and debugging of your C application.

ANIMATING C INTERPRETER™

Living C allows you to execute all your C source code on the screen. You control the code you wish to examine and the speed of animation. The cursor demonstrates exactly how your source is executing, enabling you to understand instantly how the application or prototype works – or why it doesn't!

FULL C SOURCE DEBUG

Living C conforms to the full Kernighan and Ritchie standard. Living C not only highlights all errors discovered, but also offers comprehensive error diagnostics and useful hints to solve the problem. Corrections can be made immediately, using the fully integrated Living C editor.

WINDOWS

The Living C windows allow you to constantly monitor the variables and I/O of your applications even when you are "zooming" to your next breakpoint!

I understand that I can write my applications in Living C, but what if I want to debug or modify an existing application – bearing in mind that 75% of my programming time is taken up with maintenance?

You simply compile in your C source and leave the rest to Living C! Living C not only enables you to understand how your application works but also ensures you can interpret a colleague's application and understand why it works or doesn't!! Once in Living C, the full suite of programming tools are automatically available. Combined with your rapid increase in productivity and understanding, you will now find that maintenance time is dramatically reduced.

So now I have written and tested my application, how can I use it?

You have 3 choices:

(i) Simply switch off the animation and use Living C as an interpreter

(ii) Recompile your application into your favourite C compiler (eg Microsoft, Lattice, Computer Innovations, Aztec)

(iii) Use the optional Living C code generator ($99)

What machines does Living C – Personal run on?

Living C – Personal is available for the IBM PC and all compatibles. You will need PCDOS, either twin floppy disk drives or a floppy and a hard disk with 192K RAM.

How do I order Living C – Personal?

Just fill out the coupon and send it to us along with $99 or call us on the toll free number **1 800 826-2612**

Circle no. 122 on reader service card.

Living Software, 250 North Orange Avenue, Suite 820, Orlando, Florida 32801

Living C, Living C – Personal and Animating C Interpreter are trade marks of Living Software

Circle no. 56 on reader service card.

Mail to: Living Software, 250 North Orange Avenue, Suite 820, Orlando, Florida 32801

Please Send Quantity

Living C – Personal @ $99 x ____ = $ ____

Code Generator @ $99 x ____ = $ ____

Handling & Shipping Orders $ 10.00

Florida residents please include Florida Sales Tax $ ____

TOTAL = $ ____

Payment Visa☐ MC☐ Check☐ Money Order☐

Credit Card expiry date ____ / ____

Name on Card ____

Card# ____

To order your copy of Living C-Personal, please complete this form:

Name ____

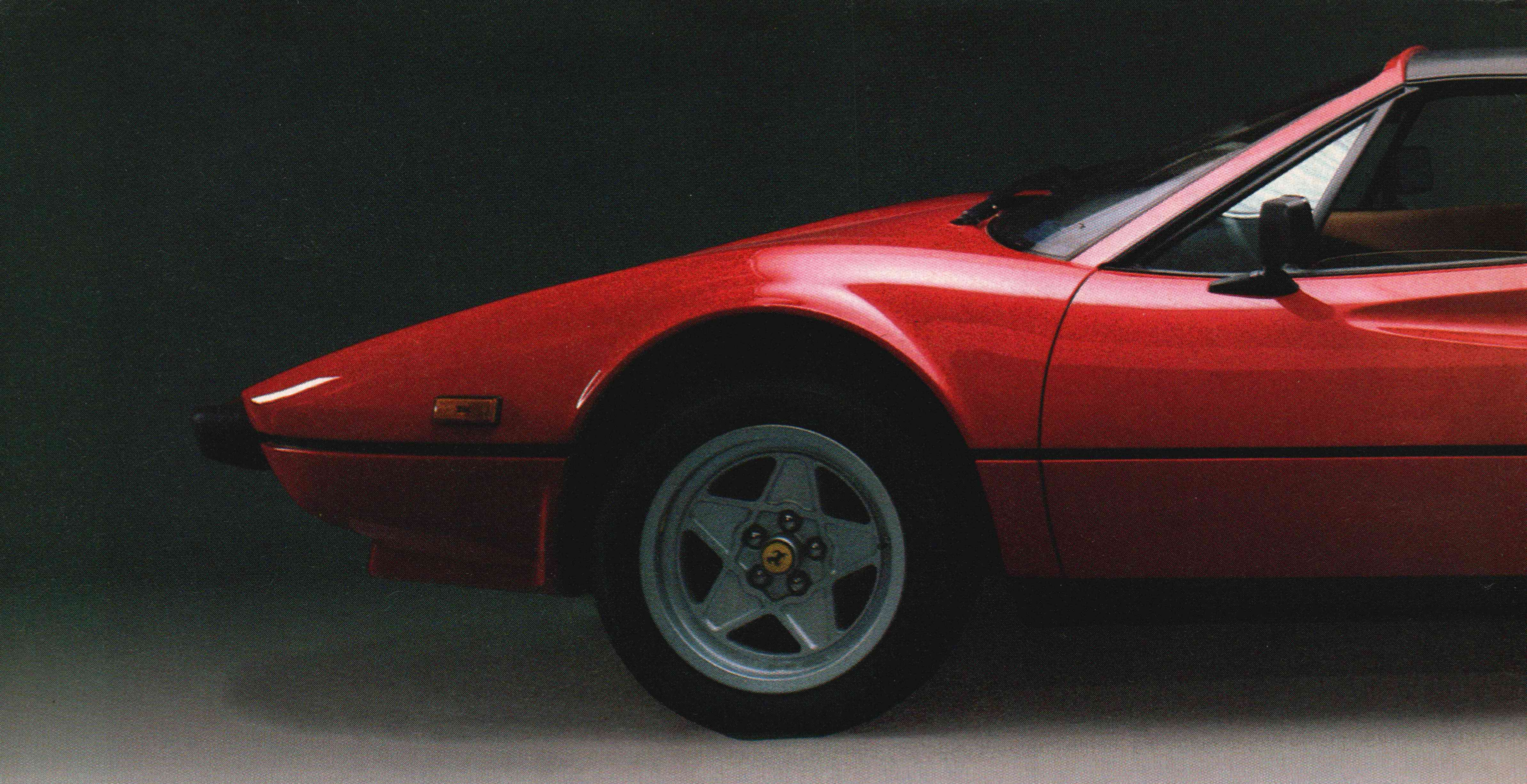
Shipping Address ____

City ____

State ____ Zip ____

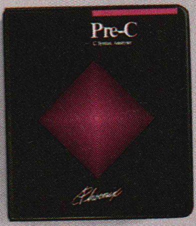
Telephone ____

Living C - Personal is available for PC-DOS with min 192KB RAM. COD's and Purchase Orders will not be accepted by Living Software.



Programmers' Pfantas

Phoenix makes programmers' dreams come true. With the best-engineered, highest performance programming tools you can find. A full line of MS-DOS®/PC DOS programs and utilities no other company offers. All designed to help you write, test and deliver the best programs possible. Top-of-the-line quality at a price you can afford. And all these products are available for the IBM® PC, XT,™ AT™ and compatibles.



Finally, A Lint For MS-DOS.

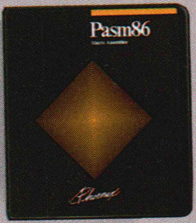
Now you can get the full range of features C programmers working in UNIX™ have come to expect from their Lint program analyzer.

With Pre-C™ you can detect structural errors in C programs five times faster than you can with a debugger. Find usage errors almost impossible to detect with a compiler. Cross-check multiple source files and parameters passed to functions. Uncover interface bugs that are difficult to isolate. All in a single pass. Capabilities no C compiler, with or without program analyzing utilities, can offer. In fact, Pre-C outlints Lint, since you can handle analyses incrementally.

Pre-C's flexible library approach lets you maintain continuity across all the programs in your shop, whether you use Pre-C's pre-built libraries, pre-existing functions you already have, or some you might want to buy yourself.

Plus, you're not limited to one particular library, and Pre-C keeps track of all the libraries you're using to make sure that code calls them correctly. **$395.**

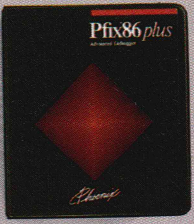
Outside the US, contact: Lifeboat Japan, Tokyo, Japan, Telex #2423296 LBJTYO, Telephone: 03-456-4101 • Repro Haganum (Stevis BV), Dinther, Netherlands, Telex #39581 REHAGNL, Telephone: 01-720-74543, 01-720-75543 • Memory Data, Sundbyberg, Sweden, Telephone: 46-8764-6700 • Roundhill Computer Systems, Marlborough, Wiltshire, UK, Telex #444453 AWARE G, Telephone: 44-467254675.



Assemble Programs Twice As Fast.

Pasm™86 will assemble MASM files two to three times faster than MASM 3.0. Pasm86 supports 8086/88, 8087, 80186 and 80286 processors.

With Pasm86's built-in defaults, you can write code quickly since you won't spend hours learning all the control statements needed at the beginning of your program. You can define symbols on the command line. Decide whether you want error messages or not. And, put local symbols within procedures. **$295.**

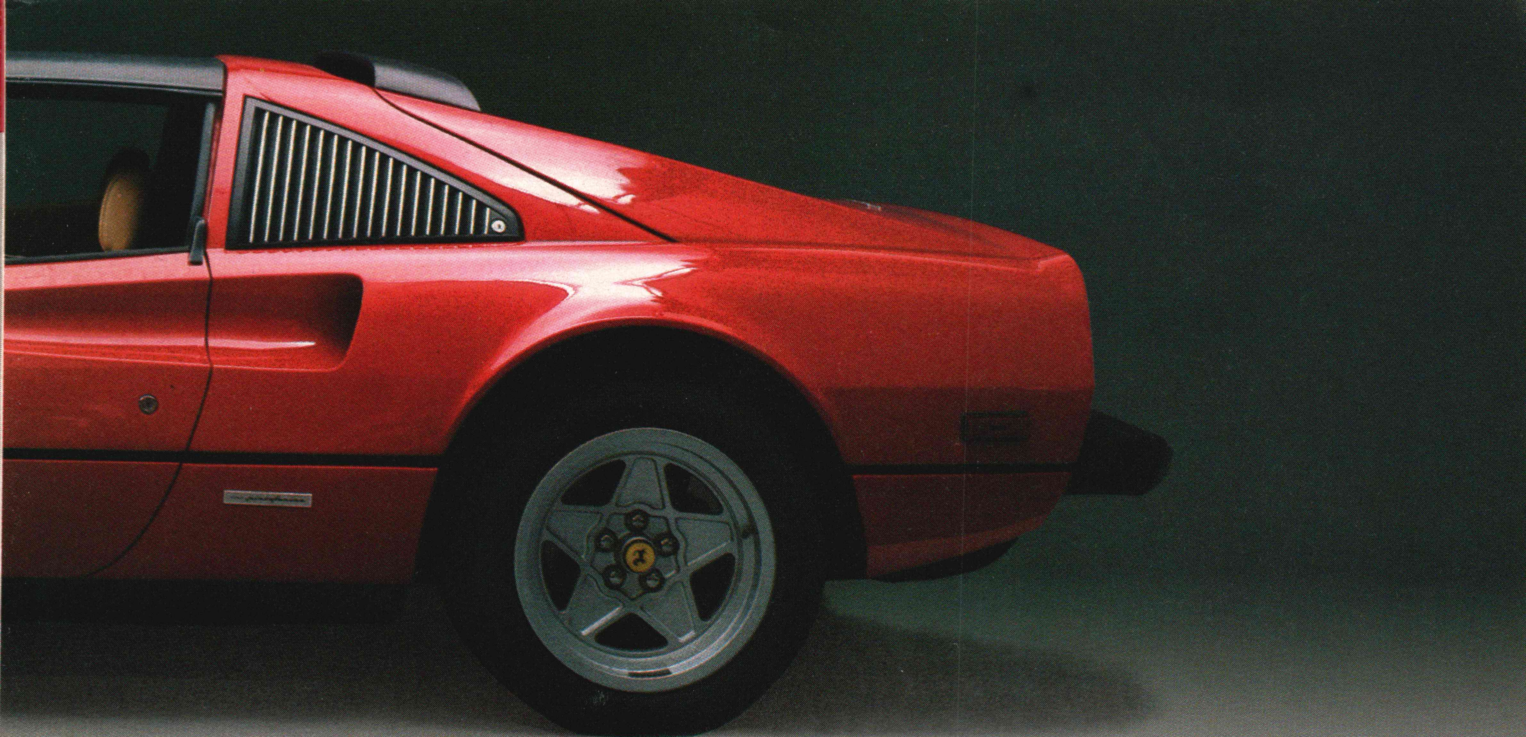


Still Fixing Bugs The Hard Way?

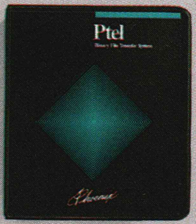
Pfix™86 Plus, the most advanced symbolic debugger on the market, eliminates the endless error searches through piles of listings. Locate instructions and data by symbolic name, using symbolic addresses. Handle larger, overlayed programs with ease.

An adjustable multiple-window display shows source, object code and data, breakpoint settings, current machine register and stack contents simultaneously. An in-line assembler allows program corrections directly in Assembly language. Powerful breakpoint features run a program full speed until a loop has been performed n times.

With a single keystroke you can trace an instruction and the action will be immediately reflected in source, object, data, stack, and register windows. Another key begins a special trace mode that executes call and loop instructions at full speed. Designed to work with both Plink™86 and MS® LINK linkage editors. **$395.**



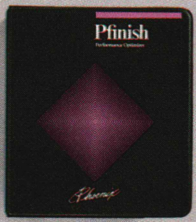
ies™ by Phoenix



Get The Lead Out Of Binary File Transfer.

Ptel™ is the universal binary file transfer program for MS-DOS 2.0 or higher. You can move binary files fast and accurately. Upload or download groups of files from Bulletin Boards or remote computers. Move files between dissimilar machines and operating systems. Ptel's advanced binary protocol, Telink!™ offers better-than- Modem7 accuracy and performance. Faster transfer speeds. An on-screen update of error correction, blocks transferred, and transmission time.

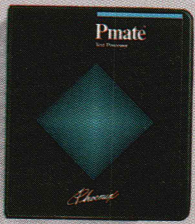
Includes popular Modem7 and XModem protocols. With checksum or CRC. Plus Kermit and ASCII. **$195.**



Maximize Your Program's Efficiency.

Pfinish™ delivers the fastest running programs possible. This performance analyzer lets you "zoom in" on the inefficient parts of your program. Whether written in Assembly language. C. PASCAL. FORTRAN. BASIC. Unlike profilers available today, Pfinish understands the structure of your program and reports the amount of activity and time spent in its subroutines or functional groups. Pfinish analyzes both overlaid and memory resident programs. Down to the instruction level. Reports are displayed. Stored on disk. Or printed out. In tabular form or histograms.

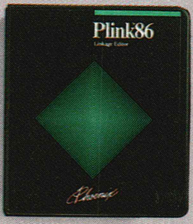
Do a dynamic program scan. Identify the most frequently executed subroutines. Find inefficient code that costs your program valuable time. Rank subroutines by execution frequency. **$395.**



Why Work With A Primitive Editor?

More than a powerful editor, Pmate™ is a text processing language. An emulator of other editors. A language-specific editor for C, PASCAL, and FORTRAN. Pmate can even run in the background!

You get full-screen, single-key editing. Ten editing buffers. Horizontal and vertical scrolling. A "garbage stack" buffer. A built-in macro language with variables, control statements, radix conversion, tracing and 120 commands that you can group and execute with a single keystroke. **$225.**



Why Squeeze Your Program More Than You Have To?

The Plink86 overlay linkage editor brings modular programming to 8086/88-based micros. Write large and complex programs without worrying about memory constraints. Work on modules individually, link them into executable files. Use the same module in different programs. Change the overlay structure of an existing program without recompiling. Use one overlay to access code and data in other overlays.

Plink86 links Intel-format modules. **$395.**

Call (800) 344-7200. In Massachusetts (617) 762-5030. Or, write.

Phoenix

Phoenix Computer Products Corp.
1420 Providence Highway Suite 115
Norwood, MA 02062

Circle **no. 47** on reader service card.

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Managing Editor *Frank DeRose*
Technical Editor *Alex Ragen*
Editorial Assistant *Sara Noah*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Allen Holub*
Copy Editor *Vince Leone*
Typesetter *Jean Aring*

Production
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Cover Artist *Tom Upton*

Circulation
Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Snee*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Kathleen Boyd*

Administration
Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*
Administrative Coordinator *Kobi Morgan*

Advertising
Advertising Director
Shawn Horst *(415) 424-0600*
Advertising Sales
Walter Andrzejewski *(617) 567-8361*
Lisa Boudreau *(415) 424-0600*
Beth Dudas *(714) 643-9439*
Michele Beaty *(317) 875-0557*
Advertising Systems Manager
Ron Copeland *(415) 424-0600*
Advertising Administrative Manager
Anna Kittleson *(415) 424-0600*

M&T Publishing, Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $46 for airmail to Canada, $62 for airmail to other countries. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank. For subscription problems, call: outside of CA 800-321-3333; within CA 619-485-9623 or 566-6947.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

People's Computer Company
Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

September 1985
Volume 10, Issue 9

CONTENTS

Editorial Calendar

The editorial calendar for the first three months of 1986 has been set. In January we will focus on programming for the 68000, in February we will present a sampling from the structured languages Pascal, Ada, and Modula 2, and finally in March we will look ahead to what computing will be like in the future, with particular concentration on parallel processing. Article submissions and new product announcements should be in our hands no later than three months in advance of any issue.

Late Breaking News

In "16-Bit Software Toolbox" in June and again this month Ray Duncan mentioned patches distributed by MicroPro Technical Support that allow WordStar to use the HP LaserJet's boldface, italics and various fonts. As this issue was going to press Ray received a letter from MicroPro stating that the LaserJet patches were intended as a temporary fix and are no longer available. Updated printer enhancement disks that include LaserJet support have been sent to all MicroPro dealers.

Corrigendum

In last month's review of C Compilers we accidentally left out the address and telephone number of Rational Systems, manufacturer of Instant C, an interpreter for the C programming language. We apologize to the folks at Rational for this omission and include their address and phone number below for the information of our readers:

Rational Systems, Inc.
P.O. Box 480
Natick, MA 01760
(617) 653-6194

This Month's Referees

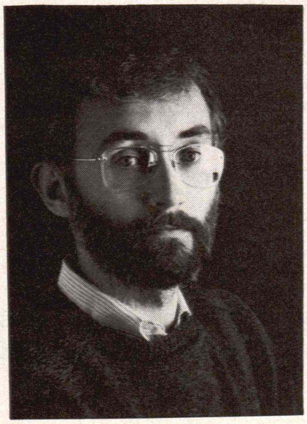
Dennis Allison, Stanford University
Allen Holub, *DDJ* Contributing Editor

Dr. Dobb's Journal

ARTICLES

COLUMNS

DEPARTMENTS



At *Micro Cornucopia*'s recent Semi-Official Gathering of hardware designers, I heard George Morrow express the computer vendor's dilemma: supplying solutions without knowing the problems. There is, Morrow explained, a very high wall. Computer hardware manufacturers dwell on one side, and in the dense jungle on the other side prowl strange questing beasts called customers. The manufacturers blindly, randomly throw machines over this wall. Occasionally one of them is rewarded with the sound of feeding somewhere beyond the wall, and they all rush to throw their machines at this latest feeding spot.

Publishing a magazine for advanced programmers is not quite such a random process: there are various means by which we can know our customers. We read what you write on the postage-paid response cards included in every issue. We do regular reader surveys, because interests can change quickly in so volatile a field as computer programming. And we have begun a spot survey that will give us quick information about our readers' interests.

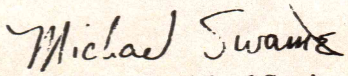
Listening to your feedback has helped us in shaping an editorial calendar for 1986. On your advice, we will be looking more closely at advanced processors in 1986, while not abandoning machines of the 8080/6502 class. We asked for feedback on our hardware issue in July, and although you were not univocal, we think we got the message. We will consequently not run a hardware issue next year, but will run hardware articles whenever they seem appropriate and good.

The trick is to satisfy the diverse interests simultaneously. As this issue exemplifies, we are focusing more on algorithms, and in other ways attempting to present tools that are as portable, as broadly useful as possible. But to follow that path to the end would mean never publishing any assembly code, and that would cut out some of the best pieces that come into the office.

So we'll continue the juggling act. We'll publish articles focusing on particular topics, presenting specialized programming tools, sometimes with code written in assembly language when that's the best medium for it, but always with tips on ways to port these tools to other environments. We'll maintain an emphasis on the underlying algorithms. And we'll try to maintain balance in the magazine, making sure that we're supplying something for everyone.

But we are still interested in good assembly language programming; in fact, our first issue of 1986 will focus on programming for the Motorola 68000 processors. It's not too late to send us your 68K code, but don't wait: the deadline for submissions is October 1.

Of course, some 68000 machines try to pass as 8088s, like the Amiga, the Mac-killer that Commodore has thrown over the wall into the IBM area, just south of a mob of hostile dealers. My friend Norm down at the Fab Lab has been researching that commonly observed phenomenon of two people achieving the same breakthrough at the same time. Norm's preliminary findings indicate that this may be part of some larger resonance in technological decision making. For example, just as IBM dropped the 5¼-inch floppy disk, Commodore picked up that medium as the means of getting software onto the Amiga quickly—IBM PC software, that is. Commodore hopes people will use the Amiga as an IBM PC with good games until the native-mode and/or graphics interface-based software arrives. If the software doesn't arrive—ah, is that why they didn't put the company name on the machine?



Michael Swaine

The C for Microcomputers

AZTEC C

PC-DOS, MS-DOS, CP/M-86, Macintosh, Amiga, Apple II, CP/M-80, Radio Shack, Commodore, XENIX, ROM, and Cross Development systems

MS-DOS, PC-DOS, CP/M-86, XENIX, 8086/80x86 ROM

Manx Aztec C86

"A compiler that has many strengths . . . quite valuable for serious work"

Computer Language review, February 1985

Great Code: Manx Aztec C86 generates fast executing compact code. The benchmark results below are from a study conducted by Manx. The Dhrystone benchmark (CACM 10/84 27:10 p1018) measures performance for a systems software instruction mix. The results are without register variables. With register variables, Manx, Microsoft, and Mark Williams run proportionately faster, Lattice and Computer Innovations show no improvement.

	Execution Time	Code Size	Compile/ Link Time
Dhrystone Benchmark			
Manx Aztec C86 3.3	34 secs	5,760	93 secs
Microsoft C 3.0	34 secs	7,146	119 secs
Optimized C86 2.20J	53 secs	11,009	172 secs
Mark Williams 2.0	56 secs	12,980	113 secs
Lattice 2.14	89 secs	20,404	117 secs

Great Features: Manx Aztec C86 is bundled with a powerful array of well documented productivity tools, library routines and features.

Optimized C compiler	Symbolic Debugger
AS86 Macro Assembler	LN86 Overlay Linker
80186/80286 Support	Librarian
8087/80287 Sensing Lib	Profiler
Extensive UNIX Library	DOS, Screen, & Graphics Lib
Large Memory Model	Intel Object Option
Z (vi) Source Editor -c	CP/M-86 Library -c
ROM Support Package -c	INTEL HEX Utility -c
Library Source Code -c	Mixed memory models -c
MAKE, DIFF, and GREP -c	Source Debugger -c
One year of updates -c	CP/M-86 Library -c

Manx offers two commercial development systems, Aztec C86-c and Aztec C86-d. Items marked -c are special features of the Aztec C86-c system.

Aztec C86-c Commercial System	**$499**
Aztec C86-d Developer's System	**$299**
Aztec C86-p Personal System	**$199**
Aztec C86-a Apprentice System	**$49**

All systems are upgradable by paying the difference in price plus $10.

Third Party Software: There are a number of high quality support packages for Manx Aztec C86 for screen management, graphics, database management, and software development.

C-tree $395	**Greenleaf $185**
PHACT $250	**PC-lint $98**
HALO $250	**Amber Windows $59**
PRE-C $395	**Windows for C $195**
WindScreen $149	**FirsTime $295**
SunScreen $99	**C Util Lib $185**
PANEL $295	**Plink-86 $395**

MACINTOSH, AMIGA, XENIX, CP/M-68K, 68k ROM

Manx Aztec C68k

"Library handling is very flexible . . . documentation is excellent . . . the shell a pleasure to work in . . . blows away the competition for pure compile speed . . . an excellent effort."

Computer Language review, April 1985

Aztec C68k is the most widely used commercial C compiler for the Macintosh. Its quality, performance, and completeness place Manx Aztec C68k in a position beyond comparison. It is available in several upgradable versions.

Optimized C	Creates Clickable Applications
Macro Assembler	Mouse Enhanced SHELL
Overlay Linker	Easy Access to Mac Toolbox
Resource Compiler	UNIX Library Functions
Debuggers	Terminal Emulator (Source)
Librarian	Clear Detailed Documentation
Source Editor	C-Stuff Library
MacRam Disk -c	UniTools (vi,make,diff,grep) -c
Library Source -c	One Year of Updates -c

Items marked -c are available only in the Manx Aztec C86-c system. Other features are in both the Aztec C86-d and Aztec C86-c systems.

Aztec C68k-c Commercial System	**$499**
Aztec C68d-d Developer's System	**$299**
Aztec C68k-p Personal System	**$199**
C-tree database (source)	**$399**
AMIGA, CP/M-68k, 68k UNIX	**call**

Apple II, Commodore, 65xx, 65C02 ROM

Manx Aztec C65

"The AZTEC C system is one of the finest software packages I have seen"

NIBBLE review, July 1984

A vast amount of business, consumer, and educational software is implemented in Manx Aztec C65. The quality and comprehensiveness of this system is competitive with 16 bit C systems. The system includes a full optimized C compiler, 6502 assembler, linkage editor, UNIX library, screen and graphics libraries, shell, and much more. The Apple II version runs under DOS 3.3, and ProDOS, Cross versions are available.

The Aztec C65-c/128 Commodore system runs under the C128 CP/M environment and generates programs for the C64, C128, and CP/M environments. Call for prices and availability of Apprentice, Personal and Developer versions for the Commodore 64 and 128 machines.

Aztec C65-c ProDOS & DOS 3.3	**$399**
Aztec C65-d Apple DOS 3.3	**$199**
Aztec C65-p Apple Personal system	**$99**
Aztec C65-a for learning C	**$49**
Aztec C65-c/128 C64, C128, CP/M	**$399**

Distribution of Manx Aztec C

In the USA, Manx Software Systems is the sole and exclusive distributor of Aztec C. Any telephone or mail order sales other than through Manx are unauthorized.

Manx Cross Development Systems

Cross developed programs are edited, compiled, assembled, and linked on one machine (the HOST) and transferred to another machine (the TARGET) for execution. This method is useful where the target machine is slower or more limited than the HOST, Manx cross compilers are used heavily to develop software for business, consumer, scientific, industrial, research, and educational applications.

HOSTS: VAX UNIX ($3000), PDP-11 UNIX ($2000), MS-DOS ($750), CP/M ($750), MACINTOSH ($750), CP/M-68k ($750), XENIX ($750).

TARGETS: MS-DOS, CP/M-86, Macintosh, CP/M-68k, CP/M-80, TRS-80 3 & 4, Apple II, Commodore C64, 8086/80x86 ROM, 68xxx ROM, 8080/8085/Z80 ROM, 65xx ROM.

The first TARGET is included in the price of the HOST system. Additional TARGETS are $300 to $500 (non VAX) or $1000 (VAX).

Call Manx for information on cross development to the 68000, 65816, Amiga, C128, CP/M-68K, VRTX, and others.

CP/M, Radio Shack, 8080/8085/Z80 ROM

Manx Aztec CII

"I've had a lot of experience with different C compilers, but the Aztec C80 Compiler and Professional Development System is the best I've seen."

80-Micro, December, 1984, John B. Harrell III

Aztec C II-c (CP/M & ROM)	**$349**
Aztec C II-d (CP/M)	**$199**
C-tree database (source)	**$399**
Aztec C80-c (TRS-80 3 & 4)	**$299**
Aztec C80-d (TRS-80 3 & 4)	**$199**

How To Become an Aztec C User

To become an Aztec C user call 1-800-221-0440 or call 1-800-832-9273 (800-TEC WARE). In NJ or outside the USA call 201-530-7997. Orders can also be telexed to 4995812.

Payment can be by check, COD, American Express, VISA, Master Card, or Net 30 to qualified customers.

Orders can also be mailed to Manx Software Systems, Box 55, Shrewsbury, NJ 07701.

How To Get More Information

To get more information on Manx Aztec C and related products, call 1-800-221-0440, or 201-530-7997, or write to Manx Software Systems.

30 Day Guarantee

Any Manx Aztec C development system can be returned within 30 days for a refund if it fails to meet your needs. The only restrictions are that the original purchase must be directly from Manx, shipped within the USA, and the package must be in resalable condition. Returned items must be received by Manx within 30 days. A small restocking fee may be required.

Discounts

There are special discounts available to professors, students, and consultants. A discount is also available on a "trade in" basis for users of competing systems. Call for information.

MANX

To order or for information call:

800-221-0440

Circle **no. 62** on reader service card.

DOBB'S POST

2 CENTS 2

Hardware vs. Software

Dear *DDJ*,

I just received the July issue of *Dr. Dobbs*. In answer to the question you pose in your editorial: No, do not include hardware articles. Other magazines, such as *Byte*, do a fine job of covering hardware. Few magazines do such a fine job of covering software.

If you expand your scope to include hardware, you will dilute your focus, and certainly reduce your value to me.

Guy Scharf
2163 Jardin Drive
Mountain View, CA 94040

Dear *DDJ*,

In answer to your editorial query about hardware articles, I'm in favor of having semi-annual hardware issues, with monographs and project papers available from you (at a nominal fee, of course). There are many of us that can't afford the big bucks for some of the hardware items, but have the skills to build them if the instructions are adequate.

Vladimir Ushakoff
4753 Preston-Fall City Rd., SE
Fall City, WA 98024-5705

Dear *DDJ*,

I have thoroughly enjoyed the hardware articles in recent issues and would like to see more of them. I had begun to think no one cared about the hardware hacker anymore. I can see your point, though, that *Dr. Dobb's* may not be the appropriate vehicle for such articles since it really is a software journal.

Why not start a separate journal, a hardware companion to *DDJ*? It could contain reviews of new chips with application examples and surveys of different classes of chips (display or disk controllers, for example) as well as construction articles. From the response you report to the recent hardware articles, I think you can see the enthusiasm there might be for such a project.

Try one issue dedicated entirely to hardware and see how it is received. You already have the best software journal. Why not have the best hardware one too?

David Nye
209 W. Lowe's Creek Rd.
Eau Claire, WI 54701

Dear *DDJ*,

This is in answer to Michael Swaine's question about coverage of hardware in *Dr. Dobb's Journal*. It seems to me that there is only one way to reply: there are already plenty of articles on hardware tinkering in other journals, and it is out of character for *DDJ* to use its limited space for such articles. An occasional, exceptional article is just tolerable. Reviews of innovative hardware, such as have appeared in *DDJ* through all its years of publication, are helpful, as even the ads are. Anything more than such as these is inappropriate and decreases the value of the journal to those of us who subscribe to and read it for what it claims to be: a journal of "Software Tools for Advanced Programmers." *Dr. Dobb's* is unique and uniquely valuable, and let's keep it that way.

In our department, we have a complete file of *DDJ*, some volumes of it (especially those with articles on Forth) getting dog-eared, and all of them well used. Of other journals, only a few back issues have been saved by our engineers, for articles on roll-your-own computers or components that they may build—someday. I would wager that this is pretty much the way it is with most *DDJ* readers.

Even in the days when many, if not most, personal computer users operated with homemade hardware from kits, *DDJ* addressed software rather than hardware needs. In these days of (almost) standardized and universally available hardware, the audience for useful software articles is larger and still growing.

Wayne C. Williams
Greenville, NC 27834

Unix Exchange

Dear *DDJ*,

I have a comment on Axel Shreiner's June 1985 Unix Exchange column on tricks with the C preprocessor. In the section on program argument standards, a set of argument parsing macros are presented after the author writes "it would be so simple to develop a standard as in" the macros. Why, I ask, is a standard being proposed that has already been implemented in a software tool that is readily available? The **getopt** command line option parser makes it simple to parse Unix command line options consistently, and unlike the proposed macros, is well accepted. Public domain versions of **getopt** have been posted to the USENET. In my opinion, the code written with **getopt** is more readable than the macros presented in the article. One reason for this might be that the macros do not behave anything like functions in the C language, and so might have unpredictable behavior.

Should you be interested, I would be happy to send a printed version of the **getopt** parser for your readers.

Gary Perlman
Wang Institute
Tyng Road
Tyngsboro, MA 01879 DDJ

NEW PERSONAL OPERATING SYSTEMS

OPERATING SYSTEM TOOLBOX $99.00

ENHANCE PRODUCTIVITY.

You'd like an operating system that meets your needs and suits your personal style. You want complete access to the full computing power of your IBM-PC, XT, or AT, with high-level functions like these:

- REAL EVENT-DRIVEN SCHEDULING
- MULTIPLE USERS PER PC
- MEMORY MANAGEMENT
- DEVICE INDEPENDENT I/O
- LOCAL AREA NETWORK SERVICES
- LOGICAL NAMES FOR SOFTWARE OBJECTS

Using Operating System Toolbox, you construct your own PERSONAL OPERATING SYSTEM in a matter of hours. We supply the complexity of a powerful operating system kernel — all you provide is a friendly user interface.

PRIORITIZED EVENT-DRIVEN SCHEDULING.

We've developed an event-driven scheduler similar to the one used by Digital Equipment Corporation in their VAX/VMS system. It runs processes based on software priorities and the occurence of system events. That means that user programs get the CPU when they need it, and relinquish it to other processes when they're waiting for resources or I/O to complete.

THE SECRET?

We've packed all of the system services, memory management, input/output, and scheduling code into a single toolbox that anyone with a C compiler and an IBM-PC, XT, or AT can use. Limited only by your creativity and imagination, just write your command language interpreter in your favorite language, like Pascal or C, and link it with the Toolbox. The resulting program is a complete operating system that will support programs written in any language. Processes execute in 4 different access modes, depending on the level of access to system data structures that is required.

OPERATING SYSTEM INTERNALS REVEALED.

Wendin supplies detailed information about how operating systems work to certified owners of Operating System Toolbox — not just how to use them, but specific algorithms and details of data structures used in the toolbox. That means that you will know the capabilities of your personal operating system, and how it works. We even give detailed descriptions of how to build a user interface and give several examples in C, Pascal, and assembly language.

INTEGRATION.

Integrate Operating System Toolbox with your application programs to make them support concurrency, advanced memory management, and input/output functions. Wendin does not charge royalties for object copies of Operating System Toolbox when integrated with your special applications.

ENHANCED SYSTEM SERVICE SUPPORT.

Personal operating systems automatically support MS-DOS and PC-DOS system calls, as well as these ENHANCED SYSTEM SERVICES

Device Dependent Input/Output Services

$ASSIGN	Assign I/O Channel to Device
$DASSGN	Deassign I/O Channel
$QIO	Queue I/O Request to Device
$QIOW	Queue I/O Request With Wait
$ALLOC	Allocate Device
$DALLOC	Deallocate Device
$GETCHN	Get I/O Channel Information
$GETDEV	Get I/O Device Information
$CANCEL	Cancel I/O Request on Channel

I/O Services for Mailboxes and Messages

$CREMBX	Create Mailbox and Assign Channel
$DELMBX	Delete Mailbox and Deassign Channel
$BRDCST	Send High-Priority Message to All
$SNDSMB	Send Message to Queue Manager
$SNDOPR	Send Message to Operator

AST (Asynchronous System Trap) Services

$SETAST	Set AST enable
$DCLAST	Declare AST

Event Flag Services

$SETEF	Set process event flag
$CLREF	Clear process event flag
$READEF	Read process event flags
$WAITFR	Wait for process event flag
$WFLOR	Wait for logical OR of event flags
$WFLAND	Wait for logical AND of event flags

Change Mode Services

$CMKRNL	Change Access Mode to KERNEL
$CMEXEC	Change Access Mode to EXECUTIVE
$CMSUPV	Change Access Mode to SUPERVISOR
$CMUSER	Change Access Mode to USER
$ADJSTK	Adjust Outer Mode Stack Pointer

Logical Name Services

$CRELOG	Create logical name
$DELLOG	Delete logical name
$TRNLOG	Translate logical name

Timer and Time Conversion Services

$GETTIM	Get System Time
$NUMTIM	Convert Binary Time to Numeric Time
$ASCTIM	Convert Binary Time to ASCII String
$BINTIM	Convert ASCII String to Binary Time
$SETIMR	Schedule delivery of AST
$SETIME	Set System Time

Memory Management Services

$EXPREG	Expand Region
$CNTREG	Contract Region
$CRETVA	Create Virtual Address Space
$DELTVA	Delete Virtual Address Space
$LKWSET	Lock Pages in Working Set
$ULWSET	Unlock Pages in Working Set
$LCKPAG	Lock Page in Memory
$ULKPAG	Unlock Page in Memory
$SETPRT	Set Protection on Pages
$SETSWM	Set Process Swap Mode

Process Control Services

$CREPRC	Create Process
$DELPRC	Delete Process
$SUSPND	Suspend Process
$RESUME	Resume Process
$HIBER	Hibernate Process
$WAKE	Wakeup Process
$SCHDWK	Schedule Wakeup
$CANWAK	Cancel Scheduled Wakeup
$EXIT	Exit Image
$LODIMG	Load Image Into Memory
$EXEIMG	Load and Execute Image
$FORCEX	Force Image Exit
$SETPRN	Set Process Name
$SETPRI	Set Process Priority
$SETRWM	Set Resource Wait Mode
$GETJPI	Get Job/Process Information

PCVMS $49.00

WHAT IS PCVMS?

PCVMS is similar to VAX/VMS, the popular operating system developed by Digital Equipment Corporation for their line of mainframe computers. Everyone thought such a sophisticated and elegant operating system could never work on a microcomputer.

A MAINFRAME OPERATING SYSTEM ON A PC.

That's when Wendin put PCVMS on IBM-PC, XT, and AT personal computers. And now, for the first time in history, you can get mainframe performance from your personal computer — at a fraction of the cost. PCVMS has the same powerful features you're used to, like MULTIPLE PROCESSES, MULTIPLE USERS, NETWORKING SOFTWARE, a RICH COMMAND SET, and a complete SET OF SYSTEM SERVICES. And it's available for less than $50, with source code included.

PCUNIX $49.00

AFFORDABLE UNIX FOR PERSONAL COMPUTERS.

PCUNIX is Wendin's implementation of the super-hit UNIX operating system developed at Bell Laboratories for PDP-11 minicomputers. Designed by systems programmers for systems programmers, a UNIX-like environment is ideal for developing system software.

A MAINFRAME OPERATING SYSTEM ON A PC.

Just when you thought UNIX was going to need an IBM-AT computer, 20 megabytes of disk space, and thousands of dollars for an AT&T license, Wendin put PCUNIX on the PC for a fraction of the cost. PCUNIX has the same powerful features you're used to, like MULTIPLE PROCESSES, MULTIPLE USERS, POWERFUL SYSTEM SERVICES, and the set of tools called the PROGRAMMER'S WORKBENCH. And it's available for less than $50, with source code included.

Don't get left behind . . . order your Personal Operating System today!

ORDER HOTLINE
509/235-8088
CREDIT CARDS WELCOME!

WENDIN ™

BOX 266
CHENEY, WA 99004

The people who make quality software tools affordable.

DR. DOBB'S CLINIC

by D. E. Cortesi

Pascal Sources

We did some reading before writing last month's diatribe on Turbo Pascal. Hating to let good research go to waste and with the hope of finding one or two compiler implementors in the readership, we thought we'd list some of the better sources. All of these are from the British journal *Software: Practice and Experience* (*SP&E*). You should be able to find back issues in a university library.

What To Optimize

A lot of data has been published on the way people use language features, with profound implications for the design of a compiler's generated code. The first paper on this subject—and one of the most-cited computer science papers ever—is D. E. Knuth's "An Empirical Study of Fortran Programs" (*SP&E*, Vol. 1, No. 2, 2nd Quarter 1970). In this pioneer study of static and dynamic usage of language features, Knuth found what many others have since verified: the most-used features are the simplest ones, and the 80–20 rule is a good approximation for dynamic execution frequency in most programs.

A more recent work in this line is M. Shimasaki, *et al.*, "An Analysis of Pascal Programs in Compiler Writing" (*SP&E*, Vol. 10, No. 2, February 1980). This team studied the static and dynamic frequency with which Pascal's features were used, examining several Pascal compilers written in Pascal. In static usage, they found that 30–40% of all statements were assignments, 30–40% were procedure calls, and most of the rest were IFs. The proportion of procedure calls was much higher than Knuth and others had found in Fortran and PL/I programs; furthermore, about half of the procedures were named only once! This may not be typical of ordinary Pascal programs, as compilers are more likely to be planned and written in top-down modular style.

When they compiled the compilers to P-code and instrumented the P-machine, they found that 40% of all executed instructions were load-to-stack operations. On breaking that down, they discovered 9% of all operations were "load constant," 19% were "load global variable," while 11% were loads from some level other than global. Of the latter, 71% (7.8% of all operations) loaded a local of the active procedure (this presumably included loading parameters). The remainder—only 3% of all operations—referred to locals of some intermediate procedure.

In "A Contextual Analysis of Pascal Programs" (*SP&E*, Vol. 12, No. 2, February 1982), R. P. Cook and I. Lee reported a static analysis of 120,000 lines of Pascal code. They produced some striking numbers: for example, that 89% of all subprograms had fewer than five local variables or that 47% of all parameters passed were constants. 99% of all subprograms had five or fewer parameters, and 92% had no more than two. These are important findings if you are designing a compiler's subroutine calling sequence. In a machine with a few registers and a stack, should you try to pass parameters in registers? Definitely yes: if you have as few as *two* scratch registers, nine subprograms in ten will receive *all* their parameters in registers!

Cook and Lee substantiated Knuth's observation that the simplest features are the most used. Among many other numbers, they reported that 94% of all array references used but a single subscript, of which 19% were written as a constant and 63% as the name of a simple variable. A compiler that, when looking at a subscript, checks for "name, delimiter" and "constant, delimiter" before going into a full expression parse is a compiler that will save time, four tries in five.

As to stack-frame usage, they found references to intermediate-level names to comprise less than 2% of all references occurring in procedures and less than 3% of those in functions. This "would seem to indicate that expensive mechanisms to implement up-level frame addressing are largely unnecessary." It would also seem to validate the design choice made in C—to have only local and global name scopes.

We can't resist pointing out that this kind of survey doesn't require deep theoretical insight or great originality. The information is genuinely significant and useful, but generating it requires only persistence, curiosity, read access to a quantity of source code, and—for dynamic studies—write access to the source code of a compiler. These things aren't rare among *DDJ* readers; if you were to study, say, the usage of C constructs in public-domain code, *DDJ* would be pleased to publish the results. And you don't have to do a broad-spectrum census like the ones described here. A survey of a single feature—the use of operators in expressions, the number of locals or parameters, even the length of names—is worth doing, especially since one such survey is likely to generate tools that can be used in others.

How to Optimize

We did run across a couple of papers that were truly original and insightful. No compiler designer should overlook them.

The first is J. Welsh's "Economic Range Checks in Pascal" (*SP&E*, Vol. 8, No. 1, January 1978). This paper doesn't deserve the obscurity into which it seems to have fallen. Welsh describes how he made a Pascal compiler use all the information available to it to identify which expressions might violate a subrange or array bound at runtime and which expressions absolutely could not. Knowing that, the compiler could generate code for runtime range checks where they were needed and omit them where they were not. The result was "highly successful" at reducing the cost of runtime checks to levels that are tolerable in production code.

The implications of Welsh's methods go well beyond range checks on array subscripts and assignments to subrange variables. After all, binary integers are only a subrange of the natural numbers, so Welsh's technique applies equally well to any integers. For example, it would allow the compiler to insert code to check for integer overflow where it might happen and omit it where it can't. Conceivably, the compiler, having forseen potential—or inevitable!—integer overflow, could try to make this impossible by rearranging an expression or converting an intermediate result to real.

Better still, the technique permits the compiler to know exactly how much binary precision is required by many integer variables and many intermediate results. Therefore, the compiler could allocate some variables as bytes even when the programmer had declared them as plain integer; or allocate byte-sized temporary variables in some expressions; or do some integer operations (especially comparisons) using faster byte-mode instructions.

Best of all, a compiler using Welsh's technique could give the programmer marvelous feedback. It could issue warning messages like: "This expression gets a runtime check because variable K might exceed a value of 135." You would look at your code and say, "Oh, so it might, I better fix that," or you might say, "No, it can't, I better declare K as a subrange instead of integer so my

intentions will be clearer."

Finally, anyone who cares about compiler design should read D. Hanson's "Simple Code Optimizations" (*SP&E*, Vol. 13, No.8, August 1983). This is a superb paper, at once practical and educational. From the abstract: "The simplicity of most programs suggests that straightforward optimizations would produce the greatest dividends. This paper describes three such optimizations suitable for one-pass compilers . . . none requires major changes to the size or structure of the compiler [but each] results in at least a 10% reduction in object code size and a corresponding reduction in execution time."

Getting Standard

If you want to write a verifiably standard Pascal compiler—or bring an old one up to snuff (hint, hint)—you could begin in worse ways than by getting in touch with the Software Consulting Services of Lehigh University (Ben Franklin Technology Center 125, Murray H. Goodman Campus, Lehigh University, Bethlehem, PA 18015). There you can obtain the latest version of the British Standards Institute's Pascal Validation Suite and Quality Control Package. The Suite contains 740 test programs. The Package consists of the Standard Pascal Model Implementation (a complete Pascal compiler in ISO standard Pascal), a program that audits any Pascal source for ISO validity (it has been applied to itself, of course), and a set of tools that let you automate your test runs under Unix.

The notice of this offer (*Sigplan Notices*, Vol. 20, No. 6, June 1985) neglects to mention price or media formats. But it exists, so you've got no excuses left, Phillipe . . . .

MSDOS Disk Speed

When we published your figures on throughput, we remarked on how unlikely some of the MSDOS numbers seemed. Users of IBM PCs and Zenith Z100s reported throughput rates, using 5-inch diskette drives, of 9 kilobytes per second and more. These were strikingly faster than the rates reported by any 8-bit system with similar disks and competitive with 8-bit systems using hard disks and M-drives.

How does MSDOS do it, we asked, and perhaps we gave the impression we doubted the numbers. (We never supposed the numbers were false; we did suspect that some unreported buffering or caching factor in MSDOS was producing apparent high speed while not doing the same amount of I/O as other systems.)

Well, we've measured the effect with our own stopwatch and it's real, alright. We still don't know how it's done, but we do know how your programs can tap into it for a major speed improvement.

Five Ways To Read

There are five distinct ways for a program to request disk input in MSDOS on an IBM PC or compatible. The modern, preferred way is to open a "handle" for the file with DOS function 3Dh then use function 3Fh to ask for input of one byte up to 64K.

Two ways are based on the CP/M-compatible functions that use a File Control Block (FCB). The program must first open an FCB with DOS function 0Fh and establish the Disk Transfer Area (buffer address) with

Tools That Make Your Job Easier

For PCDOS/MSDOS (2.0 and above/128K) • IBM PC/Compatibles, PC Jr., Tandy 1000/1200/2000, & others
For CPM80 2.2/3.0 (Z80 required/64K) • 8″ SSSD, Kaypro 2/4, Osborne I SD/DD, Apple II, & others

MIX EDITOR

Programmable, Full/Split Screen Text Processor

Introductory Offer 29.95

Great For All Languages

A general purpose text processor, the MIX Editor is packed with features that make it useful with any language. It has auto indent for structured languages like Pascal or C. It has automatic line numbering for BASIC (255 character lines). It even has fill and justify for English.

Split Screen

You can split the screen horizontally or vertically and edit two files simultaneously.

Custom Key Layouts

Commands are mapped to keys just like WordStar. If you don't like the WordStar layout, it's easy to change it. Any key can be mapped to any command. You can also define a key to generate a string of characters, great for entering keywords.

Macro Commands

The MIX Editor allows a sequence of commands to be executed with a single keystroke. You can define a complete editing operation and perform it at the touch of a key.

Custom Setup Files

Custom keyboard layouts and macro commands can be saved in setup files. You can create a different setup file for each language you use.

MSDOS Features

Execute any DOS command or run another program from inside the editor. You can even enter DOS and then return to the editor by typing exit.

MIX C COMPILER

Full K&R Standard C Language
Unix Compatible Function Library

Introductory Offer 39.95

Complete & Standard

MIX C is a complete and standard implementation of C as defined by Kernighan and Ritchie. Coupled with a Unix compatible function library, it greatly enhances your ability to write portable programs.

The Best C Manual

MIX C is complemented by a 400 page manual that includes a tutorial. It explains all the various features of the C language. You may find it more helpful than many of the books written about C.

Fast Development

MIX C includes a fast single pass compiler and an equally fast linker. Both are executed with a simple one line command. Together they make program development a quick and easy process.

Fast Execution

The programs developed with MIX C are fast. For example, the often quoted prime number benchmark executes in a very respectable 17 seconds on a standard IBM PC.

Standard Functions

In addition to the functions described by K&R, MIX C includes the more exotic functions like *setjmp* and *longjmp*. Source code is also included.

Special Functions

MIX C provides access to your machine's specific features through BDOS and BIOS functions. The CHAIN function lets you chain from one program to another. The MSDOS version even has one function that executes any DOS command string while another executes programs and returns.

Language Features

- Data Types: char, short, int, unsigned, long, float, double (MSDOS version performs BCD arithmetic on float and double-no roundoff errors)
- Data Classes: auto, static, extern, register
- Struct, Union, Bit Fields (struct assignment supported)
- Typedef, Initialization
- All operators and macro commands are supported

30 DAY MONEY BACK GUARANTEE

Orders Only: **Call Toll Free 1-800-523-9520,** (Texas only **1-800-622-4070)**

MIX Editor ____ $29.95 + shipping ($5 USA/$10 Foreign) Texas residents add 6% sales tax
MIX C ____ $39.95 + shipping ($5 USA/$25 Foreign) Texas residents add 6% sales tax

Visa ____ MasterCard ____ Card # ____________ Exp. Date ______
COD ____ Check ____ Money Order ____ Disk Format ____________
Computer ____________ Operating System: MSDOS ____ PCDOS ____ CPM80 ____
Name ____________
Street ____________
City/State/Zip ____________
Country ____________
Phone ____________

MIX software
2116 E. Arapaho
Suite 363
Richardson, Tx 75081

Dealer Inquiries Welcome
Call (214) 783-6001

D

function 1Ah. Then it can request that one record be transferred from the file to the buffer with function 14h. The size of a record is set in the FCB; it may be any number from one byte up to 64K. Alternatively, the program may use function 27h to request reading any number (from one to 64K) of records of the current size, so long as their aggregate length doesn't exceed 64K.

The fourth method of input is to use DOS interrupt (not function) 25h. This reads raw sectors unrelated to the file structure of a disk. In order to use it for file input, your program would have to interpret the FAT, disk directory, and subdirectories on its own.

The fifth way is to call on the IBM or compatible ROM BIOS through interrupt 13h. Like DOS interrupt 25h, this provides raw sector input unrelated to the file system. It's even more device oriented, however, in that you have to address data by side and cylinder, whereas the DOS interrupt uses a scheme of "logical sector numbers."

We set out to see which of these ways was the fastest and, if possible, to figure out which of them produced those high throughput numbers and why. We ran a series of experiments on a DOS 2.1 system in an XT-ized PC. We confined our recorded measurements to the double-sided diskette drive; hard disks vary widely from brand to brand, and we couldn't find out any of this one's vital statistics such as number of tracks and cylinders or seek times.

First we formatted a new diskette. On it we created a file of 16 cylinders (144K) using the BASIC program in Listing One (page 17). That file, which commenced in cylinder 1, was the data source for all experiments. The object was to see how fast it could be read.

The DOS command Copy set the initial hurdle. We read the file with

```
COPY TESTFILE.001 NUL
```

and timed it: 9.8 seconds. That's quick; just how quick wasn't clear until we tried other programs.

Measure Molasses

Our first trial was written in BASIC (see Listing Two, page 17). BASIC makes it easy to calculate a program's elapsed time. Unfortunately, we had no difficulty timing this one with a hand-held watch; it took 122 seconds to read the file. Although there's no good reason why an I/O-bound job should run any faster when compiled, we figured it couldn't very well run slower. It didn't, but at 64 seconds it still wasn't what we'd call fast.

Maybe one of the other compiled languages would do better? We compiled the program of Listing Three (page 17) with Microsoft Pascal version 3.3. It ran in 61 seconds. Then we tried the program in Listing Four (page 17), compiled by Microsoft C version 3.0. Sixty-one seconds, ho hum. Here's how four language implementations do at reading a sequential text file; the numbers are multiples of the Copy command's time:

BASICA	12.5
BASCOM	6.5
Pascal 3.3	6.2

C 3.0	6.2

You can say this makes the languages look bad, or you can say it makes Copy look extremely good. But how does Copy do it?

Down to Brass Stacks

The remaining tests had to be done at the assembly-language level. Rather than writing and assembling programs for each of many tests, we used the Assemble command of Debug to write short programs. These could be parameterized by setting counts in their registers and by modifying constants with Enter. The entry of the handle-input test, for example, resembled Listing Five (page 16). Once that code was entered, the program could be set up for a certain input length by entering a new immediate value into one instruction. For example,

```
-e 10e,48
```

sets up to read two cylinders' worth of data on each DOS call.

We took a sequence of measurements, varying the number of bytes requested per DOS interrupt. For handle input, the results were:

CX=bytes	Seconds
200h	61.8
1200h	13.6
2400h	10.2
4800h	12.0
9000h	10.2

Reading a handle by sector-sized chunks (200h bytes) is clearly not a good idea (the similarity of that time to those turned in by the C and Pascal compilers is suggestive, however). Reading by tracks (1200h bytes) is five times faster, and reading by cylinders (2400h) is faster by a remarkable factor of six. But block sizes beyond 9K, one cylinder, don't give further improvement.

The new handle operations (there must be a better name for them) are supposed to replace the older CP/M-like ones. To do that, they had better be no slower, and our tests indicate they are not. First we tried reading single records with function 0fh while varying the size of the record:

Record	Seconds
80h	64.5
1200h	13.8
2400h	10.0
4800h	12.0
9000h	10.0

The results are practically identical to those for function 3Fh above—differences of 0.2 second aren't significant for hand-made timings. The time when reading 128-byte records is suggestively close to the BASCOM time.

When we held the record size constant at 200h and read different numbers of records with function 27h, we got identical times. The lesson is that no matter which DOS function you use, you should request at least 4,608 (nine times 512) bytes at a time. To ask for less is to multiply your diskette input time by a factor of 3 to 6.

Greased Lightning

An assembly program reading full cylinders can approach the speed of the Copy command. Is that the limit? We suspected as much but continued our experiments.

When we used the ROM BIOS interrupt 13h to read the same cylinders, the best time we could achieve was 13 seconds, 30% *longer* than the best time we got with DOS input requests! Now we know two things: the DOS BIOS doesn't use the ROM BIOS for its disk input, and DOS's BIOS does it *better*.

That made it mandatory to try out DOS interrupt 25h, presumably the foundation of the DOS file services. Here are the results:

CX=sectors	Seconds
1h	56.0
2h	31.0
4h	19.6
8h	12.5
9h	6.8
12h	6.8
24h	10.2
48h	8.5

Now, ponder those numbers a moment. The best times, as before, occur when the program asks for one track or one cylinder per call. Our program set up by forcing the disk to cylinder 0; during the timed portion, it read cylinders 1–16, inclusive. That's 32 tracks read plus 16 one-track seeks. It takes 200 milliseconds for a 5-inch diskette drive to rotate once. There are just 34 periods of 200-milliseconds in 6.8 seconds. Therefore, DOS sequential input can reach and sustain an input rate of one track per disk rotation!

Hurrah for DOS; no faster input rate is possible. But we are left with some mysteries. What does DOS do differently from the ROM BIOS to get such performance out of the diskette adaptor? Why is 4-cylinder input slower than 2-cylinder; why is 3-cylinder input slower still? How does the Copy command interface to DOS? It consistently shaved a couple of tenths of a second off the best time we could get out of handle or FCB input.

To summarize these results formally: for $1 \leqq n \leqq 8$, if a program requests $n+1$, 512-byte sectors on each input operation, its input time will be roughly proportional to $1/n$. We can see the practical difficulties of giving every open file a 4,608-byte buffer, but still, isn't it sad how slow your compiled programs run compared to the Copy command?

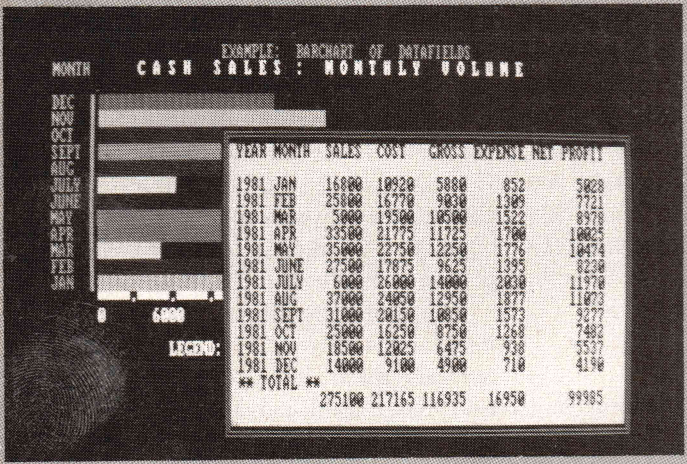
DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

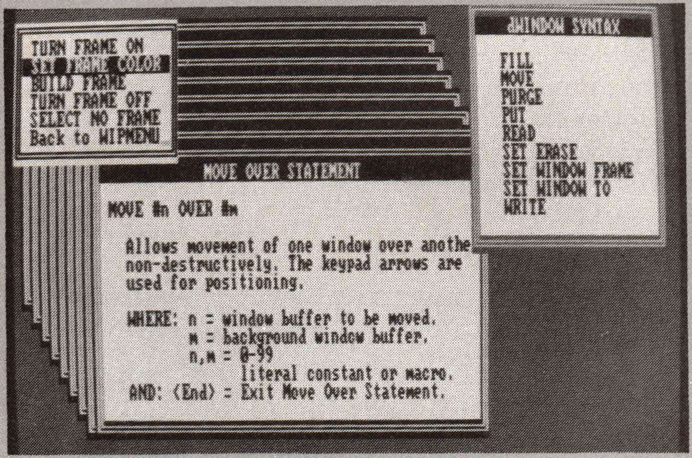
dWINDOW©!

An assembly language (9K) program adding **25 NEW WINDOWING SYNTAX STATEMENTS** to dBASE II, featuring:

- In-memory and disk-based window files
- Speeds up application program execution time
- Fast video response time
- Reduces application code requirements
- Fully compatible with dBASE II .PRG files
- Full screen, free text input with TYPE command



- Character Graphics / Modem compatible
- User Friendly MENU GENERATION (WIP.PRG)

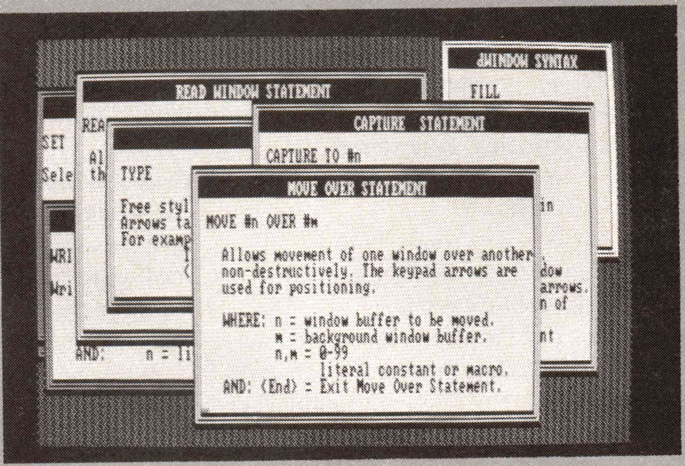


- READ WINDOW menu selection available
- Executable DOS command at dot prompt or under .PRG control
- INKEY statement allows keyboard entry of full ASCII set
- All 256 characters and color attributes selectable for both mono and color systems. (Bit-mapped graphics card or mouse not necessary.)
- PRINT SCREEN statement
- CAPTURE – full or partial screen to window
- SET WINDOW PATH – disk-based, tree-structured window files
- SET SYNC on/off – VIDEO adjustment of snow and glitches
- 100 dynamically allocated memory window buffers allowable

- Overlapping, moving windows
- Read current CURSOR position
- Select ERASE attribute (All shades and colors)
- User defined windows; WINDOW STRUCTURE programmable
- Change background / foreground colors easily

Improvements to dBASE II:

- Line 0 made available for @ SAY or GET statements
- Line 25 available for @ SAY statements
- Split-Screen MODIFY COMMAND
- *** SYNTAX ERROR *** processing compatibility



Training and Learning tools:

- on-line, menu-driven HELP
- 66 Window File examples
- 33 dBASE II .PRG files — sample applications using dWINDOW syntax statements and algorithms, including:
  - The IDEA . . . A menu-driven guide to dWINDOW
  - WIP . . . WINDOW INFORMATION PROCESSING and GENERATION
  - BARCHART . . . Columnar Report Form with Barcharts
  - NOTEPAD . . . On-line database of Quick Notes
  - ZOOM . . . Automatic program generation; scaled windows
  - PAINT – Graphic art controlled with keypad arrows

System Requirements:

- IBM PC, XT, or AT (and compatibles) with 192K
- DOS version 2.0
- MONOCHROME or Color display
- Assembler Language
- dBASE II version 2.4 or 2.41

Price: $165.00 plus $4.00 (shipping)

To order, phone (503) 221-1806 or write:

Liberty Bell Publishing 4640 S.W. Macadam, Suite 150 Portland, Oregon 97201

Circle no. **60** on reader service card.

Dr. Dobb's Clinic (Text begins on page 10)

Listing One

```
100 'Create a 16-cylinder file on a d/s, 5-inch diskette
110 OPEN "TESTFILE.001" FOR OUTPUT AS #1
120 LET D$ = SPACE$(254) :' plus CR, LF = 256 bytes
130 FOR CYL = 1 TO 16
140     FOR SIDE = 0 TO 1
150         FOR SECTOR = 1 TO 9 :' DOS 2.x and above
160             FOR BYTES = 1 TO 512 STEP 256
170                 PRINT#1,D$
180             NEXT BYTES
190         NEXT SECTOR
200     NEXT SIDE
210 NEXT CYL
220 CLOSE #1
```

End Listing One

Listing Two

```
100 'Read a file and see how long it takes
110 ' -- function to convert TIME$ to seconds
120 DEF FNSEC(X$) = 3600 * VAL(LEFT$(X$,2)) +
60 * VAL(MID$(X$,4,2)) +
VAL(RIGHT$(X$,2))
130 OPEN "TESTFILE.001" FOR INPUT AS #1
140 STARTS$ = TIME$
150 WHILE NOT EOF(1)
160     LINE INPUT#1,D$
170 WEND
180 STOPS$ = TIME$
190 PRINT FNSEC(STOPS$)-FNSEC(STARTS$);" seconds."
```

End Listing Two

Listing Three

```
program readtest(output);
var
        fin : text;
        dat : string(255);
begin
        assign(fin,'TESTFILE.001');
        reset(fin);
        writeln('start timing',chr(7));
        while not eof(fin) do
                readln(fin,dat);
        writeln('stop timing',chr(7));
end.
```

End Listing Three

Listing Four

```
#include <stdio.h>
main()
{
        FILE *fin;
        char dat[256];
        if (NULL != (fin = fopen("TESTFILE.001","r")))
        {
                fputs("start timing \x7\n",stderr);
                while(NULL != fgets(dat,255,fin)) ;
                fputs("stop timing \x7\n",stderr);
        }
        else
                fputs("can't open file\n",stderr);
}
```

End Listing Four

Listing Five

```
C>debug
-f 0 Lffff 0
-e 80 'A:TESTFILE.001',0
-a 100
0C44:0100 mov dx,80 ; filespec string
0C44:0103 mov ax,3d00 ; open it
0C44:0106 int 21
0C44:0108 mov bx,ax ; handle in BX
0C44:010A mov dx,200 ; data to DS:200
0C44:010D mov cx,200 ; block size
0C44:0110 mov ax,3f00 ; read CX bytes
0C44:0113 int 21
0C44:0115 cmp ax,cx ; read ok?
0C44:0117 jz 110 ; if so repeat
0C44:0119 nop ; breakpoint here
0C44:011A mov ax,3e00 ; close BX=hdl
0C44:011D int 21
0C44:011F jmp 100 ; set up for next
0C44:0121 ^C
```

End Listings

C CHEST

How Compilers Work—A Simple Desk Calculator

by Allen Holub

Over the years *DDJ* has published several recursive descent compilers, most notable among which are the various versions of Small-C. This month we're going to talk about how this sort of compiler works. In the interest of clarity we'll reduce the problem from recognizing a real computer language to analyzing a small arithmetic expression, a function that is part of a desk calculator program. The same techniques are applicable to both. Our expression analyzer will take as input an ASCII string representing an arithmetic expression. Only numbers, parentheses, and the operators **+ – /** and ***** are legal. The analyzer will return the result of the evaluation. For example, if you give it the string **(3+1)*2** the analyzer will return the number **8**. The routine is pretty stupid, but the purpose of this exercise is to understand compilers, not to analyze complicated expressions.

Every compiler has three functionally distinct parts. These parts are often combined, but it's best to look at them as separate functions. The first part of the compiler is a "token" recognizer. A token is some collection of ASCII characters from the input stream that are meaningful to the compiler when taken as a group. That is, a program can be seen as a collection of tokens, each of which is made up of one or more sequential ASCII characters. For example, the ASCII character **;** is a token in C, similarly the keyword **while** is a token. The matter is complicated by operators like **+**, **++** and **+=**, all of which are single tokens. A token is, then, a sort of programming atom, an indivisible part of the language (you cannot, for example, say **wh ile**) and a token recognizer is a subroutine that, when called, will return the next token from the input stream. Usually tokens are represented internally as an enumerated type or as a set of integer values corresponding to **#defines** in a header file somewhere, and the token recognizer returns this integer value. (Small-C doesn't do this. Rather, various subroutines within the compiler retrieve the tokens from input one at a time as they are needed.)

The second part of the compiler, and the part that does most of the work, is the "parser." The verb to parse retains its meaning when applied to compilers: "to resolve (a sentence, etc.) into its component parts of speech and describe them grammatically."[1] Just replace the word "sentence" with "program." Computer languages may be described by means of a formal grammar (we'll look at these in a moment) and the parser breaks up a program into its component parts and interprets the parts in a larger, grammatical context. That is, a parser organizes the tokens returned from the token recognizer in such a way that the compiler can generate code conveniently.

A good (though not very practical) way to look at the process is as the creation of a "parse tree." For example, The expression **(a – b)*(c – d)** can be organized into the tree shown in Figure 1 (page 20).

The third part of the compiler, the code generator, traverses the parse tree in an orderly way, generating code according to certain rules. For example, if we do a "post order" traversal of the tree shown in Figure 1 (visit the left node, the middle node, the right node and then the root recursively) the tokens will be read in the following order:

```
( a b – ) ( c d – ) *
```

Now, the expression may be evaluated by applying the following rules to each token in the tree as it is visited:

1) if the token is a parenthesis, do nothing;
2) if the token is a variable (**a**,**b**,**c** or **d**), push the variable onto a stack;
3) if the token is a minus (**–**), pop two items off the stack, subtract them and push the result;
4) if the token is an asterisk (*****), pop two items off the stack, multiply them together and push the result.

When you're done parsing (traversing the tree), the answer will be on the top of the stack. Owners of Hewlett Packard calculators will be familiar with the process. In a real compiler, the actual rule applied will be some function of the type of token found and the position of that token in the parse tree.

There are several flavors of parsers.[2] Most compilers use table driven parsers for several reasons. It's easier to automate compiler creation with table driven parsers; they're also more efficient. The Unix utility YACC (Yet Another Compiler Compiler), when given a formal description of a programming language, creates a set of tables that can be used by a generic, table driven parser. Similarly, LEX (LEXical analyzer) can output a C program that recognizes tokens.[3]

Unfortunately, most public domain compilers (and many commercial ones) don't use the more sophisticated table driven methods (Small-C is no exception). These compilers use a parsing method known as "recursive descent." Recursive descent parsers are easier to understand than their table driven cousins. However, they have disadvantages. They are inherently inefficient, using large amounts of stack space, and con-

struction of them cannot be automated. To change the way a recursive decent compiler works, you have to change the compiler itself. To change a table driven compiler you need only change the table. So, maintenance is a problem with recursive descent compilers.

Grammars: Representing Computer Languages

The best way to start writing a program is to reduce the problem to some sort of symbolic form. Pseudocode, flow charts, Warnier-Orr diagrams are all examples of this kind of symbolic reduction. Compilers are no exception to this process. When writing a compiler, you start by representing the programming language to be compiled in a formal, symbolic format called a "grammar." Any one programming language can be described by several grammars. The type of parser you're going to use will determine which of these is best for your application. The most useful notation used for grammars is the Backus-Naur Format (abbreviated BNF), which we'll see in a moment.

In order to create our expression analyzer, we need to start with a grammar. The first question to ask is: what exactly is an expression? You'll remember from high school that an expression is composed of factors. A factor by itself (i.e. a single number) is an expression, as are two factors separated by an operator. BNF representations of these two rules are:

<expression> ::= <factor>
<expression> ::=
<factor> <operator> <factor>

We can save some typing by using the vertical bar to represent **OR**:

<expression> ::= <factor>
| <factor> <operator> <factor>

You'll note in these definitions that no element of the BNF definition of the expression is a real symbol, one that can actually be found in the input stream. That is, <factor> and <operator> both have to be defined further before they can be related to a real program. Symbols, such as <factor>, which need further definition, are called "non-terminal" symbols. Symbols that *can* be found in the input are called "terminal" symbols. The four terminal symbols that can be operators are **+ – /** and *****. A BNF rule for <operator> is:

<operator> ::= + | – | * | /

Defining a factor is a little harder. A factor can be a number but it can also be another expression (as in: **a+b–d**, **a+b** is one factor and **d** is the second factor). A BNF definition of factor is:

<factor> ::= <number>
| <expression>

The only symbol yet to be defined is <number>. Since a number is an easy thing for the token recognizer to find, we'll cheat a little and just define <number> in English. Our entire grammar is shown in Figure 2 (at

right).

Notice that every non-terminal used to the right of a ::= is also defined to the left and that no terminal symbols are found to the left of a ::=. Let's test the grammar by plugging in an example: **1+2**. **1** and **2** are both <number>s so we can replace them with the equivalent non-terminal symbols by using rule 4:

1 + 2
<number> **+** <number>

According to rule 3, a single number is also a factor, so we can do another replacement:

<number> **+** <number>
<factor> **+** <factor>

The **+** can be evaluated using rule 2:

<factor> **+** <factor>
<factor> <operator> <factor>

And finally, by using rule 1, we can replace the above with a single <expression>:

<factor> <operator> <factor>
<expression>

So, we can reduce the input tokens **1+2** to an <expression> using the rules of the grammar. Therefore, we conclude that **1+2** is a legal expression in our grammar.

What if there's an error in the expression? Let's try to parse **1+***. We can apply rules 3 and 6 to yield:

<number><operator><operator>

and then apply rule 4 to get:

<factor><operator><operator>

But there is no rule we can apply to reduce this any further. So, we conclude that **1+*** is not an expression as defined by our grammar.

Parsing with a Grammar

So a parser can be seen as a program that reduces a collection of input tokens to a single non-terminal. We have just "parsed" the expression **1+2**. To turn a parser into a real com-

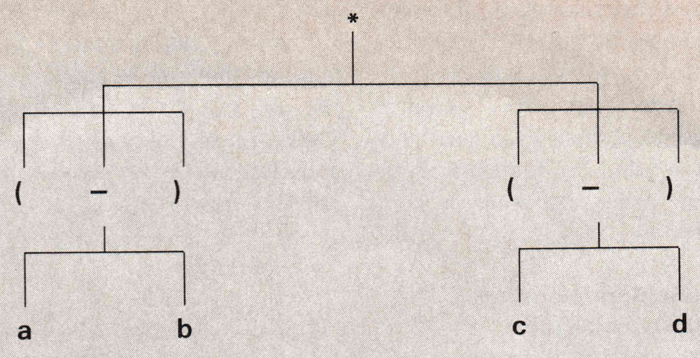


Figure 1
Parse Tree for the Expression (a – b) * (c – d)

1) <expression>	::=	<factor>
	¦	<factor> <operator> <factor>
2) <operator>	::=	+ ¦ – ¦ * ¦ /
3) <factor>	::=	<number> ¦ <expression>
4) <number>	::=	A string of ASCII characters in the range '**0**' to '**9**'

Figure 2
A Simple Expression Recognition Grammar

Grammar:

1) <expression>	::=	<factor>
2) <expression>	::=	<factor> <operator> <factor>
3) <operator>	::=	+ ¦ – ¦ * ¦ /
4) <factor>	::=	<number>
5) <factor>	::=	<expression>
6) <number>	::=	Any string of ASCII characters in the range '**0**' to '**9**'.

Action rules:

1) Do nothing.
2) Pop two objects off the stack, apply the operator remembered in rule 3 and then push the result.
3) Remember the operator for rule 2.
4) Push the number onto the stack.
5) Do nothing.
6) Translate the ASCII string into a number.

Figure 3
Adding Actions to the Grammar

			rule:	**action:**
1	**+**	**2**		
<number>	**+**	**2**	6	Translate ASCII to int
<factor>	**+**	**2**	4	Push 1
<factor>	**+**	<number>	6	Translate ASCII to int
<factor>	**+**	<factor>	4	Push 2
<factor>	<operator>	<factor>	3	Remember the +
<expression>			2	Pop the 1 & 2, apply + and push the result.

Figure 4
Parsing 1 + 2 Using Grammar in Figure Three

	Apply rule:	Action:
1 + 2 − 3	—	Start here
<number> + 2 − 3	6	Translate "1" to int
<factor> + 2 − 3	4	Push 1
<factor> <operator> 2 − 3	3	Remember +
<factor> <operator> <number> − 3	6	Translate "2" to int
<factor> <operator> <factor> − 3	4	Push 2
<expression> − 3	2	Pop two numbers, apply + and push result.
<factor> − 3	1	Do nothing.
<factor> <operator> 3	3	Remember −
<factor> <operator> <number>	6	Translate "3" to int
<factor> <operator> <factor>	4	Push 3
<expression>	2	Pop two numbers off the stack, subtract and push the result.

Figure 5
Parsing 1 + 2 − 3

<expr>	::=	<factor>	(1)
	|	<factor> * <expr>	(2)
	|	<factor> / <expr>	(3)
	|	<factor> + <expr>	(4)
	|	<factor> − <expr>	(5)
<factor>	::=	(<expr>)	(6)
	|	− (<expr>)	(7)
	|	<constant>	(8)
	|	− <constant>	(9)
<constant>	::=	A string of ASCII characters in the range **'0'–'9'**.	

Figure 6
A More Realistic Expression Recognizing Grammar

piler, we need to make it do something active too, namely, generate code. So, we associate an action rule with each grammatical rule. Our grammar, slightly shuffled around and with action rules added is shown in Figure 3 (page 20). Every time we apply a grammatical rule, we'll also perform the action specified in the equivalent action rule. **1+2**, parsed with the grammar in Figure 3 is shown in Figure 4 (page 20). A somewhat more involved example is given in Figure 5 (at left). You're beginning to see how the process works. If the action rules had generated the code necessary to perform the operation, rather than actually doing the operation itself, we'd have a compiler.

As you've probably noticed, the grammar just defined isn't very useful. At very least we'd like to have parentheses and negative numbers. We'd also like to be able to negate an entire expression (i.e. **−(17*11)**). You'll also notice in the above examples that the expression is just parsed left to right, with all possible substitutions made as we parse. A more realistic grammar performs its substitutions in a somewhat more complex way, and the grammar has to reflect this complexity. In addition, a grammar has to be organized so that the parser can always tell what rule to apply based on the current input symbol and the current rule being processed. A better expression recognizing grammar is given in Figure 6 (at left). We'll use this grammar in our actual program.[4]

A Recursive Descent Parser

The best way to see how a parser works is to look at one. Before discussing the parser proper, I want to talk a little about how the program (Listing, page 25) is organized. The actual subroutines in the parser are highly recursive. As such, they'll use up a lot of stack space as they work. Because of this stack usage, we want to pass as few parameters as possible to the subroutines (because all these parameters take up stack space). So, we make global those variables that would normally be passed to the subroutines as arguments. However, this practice, introduces new problems. In

C, all non-static global variables are shared between all modules in a program. But the expression parser is probably going to be a library routine and we don't want it to interfere with the normal workings of the rest of a program. Moreover, we don't want the programmer to have to remember that certain globals are used by a particular library routine and can't be used anywhere else. So, we make the globals static. We'll also make static those subroutines that are only used internally. Now, however, we need some way to initialize the static globals from outside the parser module. We do this initialization with the "access routine" starting on line 84 of the Listing (the only externally accessible subroutine in the module). This access routine (called **parse( )**) does nothing but initialize our globals and then call **expr( )** to do the work.

Another organizational concern is the **main( )** routine on lines 34–79. The primary purpose of **main( )** is to test **parse( )**, thus the **#ifdef/#endif** on lines 32 and 81. **DEBUG** is not **#defined** when we compile for inclusion in a library. The **main( )** routine given is moderately useful in its own right. You can enter the expression (**17/(2*12)**) from the command line or you can just type **expr** and then enter expressions as the program prompts you, sort of a rudimentary desk calculator.

Moving back to parsers, there are a few things to notice about the grammar in Figure 6. First, the left-most symbol following the ::= is always either a terminal or the same non-terminal for all rules. That is, all rules associated with <expr> have <factor> as their left-most symbol. The left-most symbol of all <factor> rules is either a terminal (**(** or **−**) or the non-terminal <constant>. The left-most symbol of a constant has to be an ASCII digit. This property of the grammar is required by the parser so that it can know what rule to apply in a given situation. For example, when evaluating an <expr>, the parser will always apply a rule associated with <factor> first.

A second property of the grammar is that the definitions for <expr> and <factor> are recursive. An <expr> is defined in terms of other <expr>s. The recursion in <factor> is two levels deep. A <factor> is defined in terms of an <expr>, which is in turn defined in terms of a <factor>. The recursion in the grammar suggests that we can also use recursion in a parser that implements the grammar.

So, given an appropriate grammar, we can translate that grammar directly into a parser. In the program given here, all non-terminal symbols in the grammar have an equivalent subroutine with the same name. The routine for <expr> starts on line 104, for <factor> on line 124 and for <constant> on line 155.

Looking again at the grammar in Figure 6, we see that the first thing done in all the <expr> rules (1–5). is to look for a <factor>. Similarly, the first thing the subroutine **expr( )** does is call the subroutine **factor( )** (on line 108). Looking back at the grammar, the next thing <expr> does is look for a terminal symbol (either a * / + − or a null string). The equivalent code is the switch on lines 110–117. The default case takes care

of the null terminal (rule 1). The recursive evaluation of <expr> in rules 2–5 is also done in the switch. On line 119 **expr( )** returns the evaluated expression.

Factor( ) is somewhat more complex. It first checks (on lines 128–132) for the leading minus sign required by rules 7 and 9. After stripping off the minus, rules 6 and 7 become identical, similarly rules 8 and 9 are identical once the minus is gone. So, **factor( )** now decides which rule to process by looking for a leading **(** (on line 134). If it doesn't find the parenthesis, rule 8 is processed (line 135) by calling the subroutine **constant( )**, otherwise rule 6 is processed by skipping past the parenthesis and then calling **expr( )** (lines 138–139). We can also do some error checking here by looking for a close parenthesis when **expr( )** returns (lines 143–147).

The final part of the parser is the routine **constant( )** on lines 155–169. This routine is essentially **atoi( )**, however it advances the string pointer past the end of a number and flags an error if a number isn't found.

You'll note that in this program (and in the Small-C Compiler) the three functional parts of the compiler are merged together. There is no explicit token recognizer, rather each routine is responsible for advancing the global string pointer (**Str**) past the token being processed. Similarly, the code generation part of the compiler is integrated into the parser. In our example, code generation is replaced by the various return statements. In a real compiler the routine **factor( )** would generate code to push a value onto a run-time stack rather than return a value. The switch on lines 110 to 117 would be replaced by something like:

```
switch( *Str )
{
case '+':
        Str++;
        expr( );
        codegen(1);
        break;
case '-':
        Str++;
        expr( );
```

```
        codegen(2);
        break;
    /* etc. */
    }
```

Since the code to push one number onto the stack is generated in **factor( )**, The first number will be pushed by the **factor( )** call on line 108. By the time **expr( )** returns, the code to push the second number will have been generated (by the **factor( )** call inside **expr( )**). The call to **codegen(1)** inside the switch generates the code needed to pop two numbers off the stack, add them together, and push the result. The **codegen(2)** call behaves similarly, but it subtracts rather than adds.

So, that's the bulk of the problem. Hopefully a better understanding of what's going on will help when you try to sort out the workings of compilers such as Small-C.

Footnotes

[1] *The Compact Edition of the Oxford English Dictionary* (Oxford: Oxford University Press, 1971) p. 2083.

[2] A good, short, description of table-driven parsing techniques can be found in: Dr. Henry A. Seymour, "An Introduction to Parsing," *Dr. Dobb's Journal*, 98 (December, 1984), pp 78–86. A more in-depth look at the subject, and at compiler design in general, can be found in: Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Reading, Addison-Wesley, 1979; P. M. Lewis, D. J. Rosenkrantz and R. E. Stearns, *Compiler Design Theory* (Reading: Addison-Wesley, 1976).

[3] Axel T. Schreiner and H. George Friedman, Jr., *Introduction to Compiler Construction with Unix*, (Englewood Cliffs: Prentice-Hall, 1985) is the best guide to YACC that I know of. It takes you, step by step, through the entire process of generating a C compiler using YACC and LEX. More terse descriptions of both programs are in: Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," *Unix Programmer's Manual* Vol. 2 (New York: Holt, Rinehart and Winston, 1979) pp. 353–387 and "Lex—A Lexical Analyzer Generator," *ibid.*, pp. 388–400.

[4] A grammar for the C language is in Kernighan & Ritchie, *The C Programming Language* (Englewood Cliffs: Prentice-Hall) 1978 p. 214–219.

DDJ

C Chest Listing

(Text begins on page 18)

```
#include <stdio.h>

/*  EXPR.C:  (C) Copyright 1985, Allen I. Holub.  All rights reserved
 *
 *      Evaluate an expression pointed to by str. Expressions evaluate
 *      right to left unless parentheses are present. Valid operators
 *      are * + - / for multiply add, subtract and divide. The expres-
 *      sion must be formed from the following character set:
 *      {  0123456789+-*()/  }. White space is not allowed.
 *
 *       <expr>     ::=     <factor>
 *                        | <factor> * <expr>
 *                        | <factor> / <expr>
 *                        | <factor> + <expr>
 *                        | <factor> - <expr>
 *
 *       <factor>   ::=       ( <expr> )
 *                        | -( <expr> )
 *                        |  <constant>
 *                        | -<constant>
 *
 *       <constant> ::=  A string of ASCII chars in the range '0'-'9'.
 *
 *-------------------------------------------------------------------
 *  Global variables:
 */

static  char    *Str  ;   /* Current position in string being parsed  */
static  int     Error ;   /* # of errors found so far                 */

/*------------------------------------------------------------------*/
#ifdef DEBUG

main(argc, argv)
char     **argv;
{
```

(Continued on next page)

 (Listing Continued, text begins on page 18)

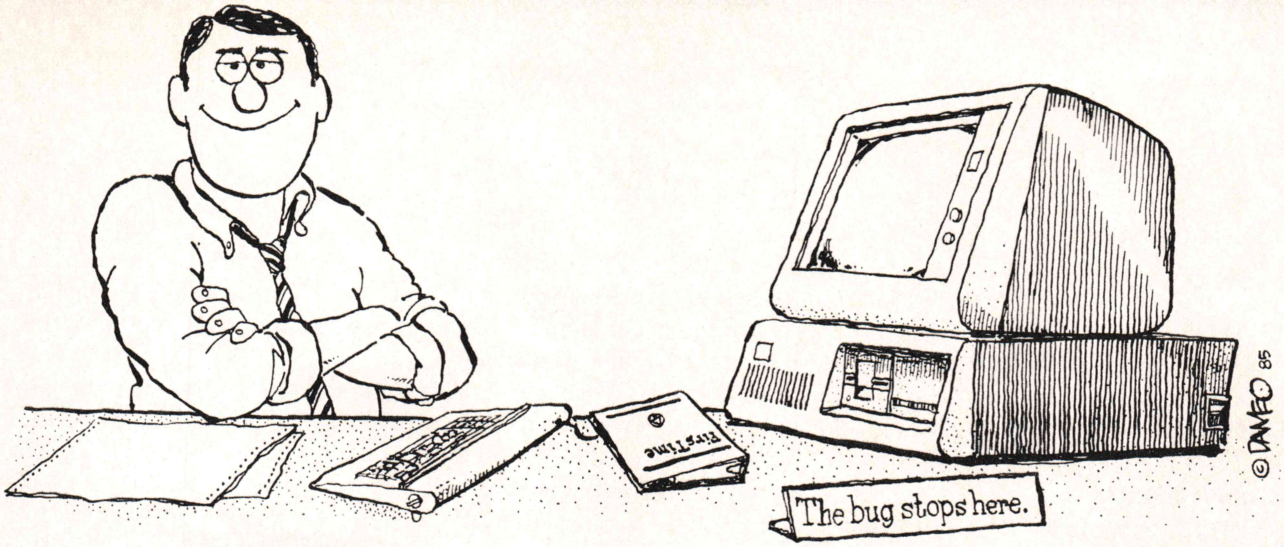
```
        /*      Routine to exercise the expression parser. If an
         *      expression is given on the command line it is
         *      evaluated and the result is printed, otherwise
         *      expressions are fetched from stdin (one per line)
         *      and evaluated. The program will return -1 to the
         *      shell on a syntax error, 0 if it's in interactive
         *      mode, otherwise it returns the result of the
         *      evaluation.
         */

        char buf[133], *bp = buf ;
        int err, rval;

        if( argc > 2 )
        {
                fprintf(stderr, "Usage:  expr [<expression>]");
                exit( -1 );
        }

        if( argc > 1 )
        {
                rval = parse( argv[1], &err );
                printf(err ? "*** ERROR ***" : "%d", rval );
                exit( rval );
        }

        printf("Enter expression or <CR> to exit program\n");

        while( 1 )
        {
                printf("? ");

                if( gets(buf) == NULL  ||  !*buf )
                        exit(0);

                rval = parse(buf, &err);

                if( err )
                        printf("*** ERROR ***\n");
                else
                        printf("%s = %d\n", buf, rval);
        }
}

#endif
/*-----------------------------------------------------------------*/

int     parse( expression, err )
char    *expression;
int         *err;
{
        /* Return the value of "expression" or 0 if any errors were
         * found in the string. "*Err" is set to the number of errors.
         * "Parse" is the "access routine" for expr(). By using it you
         * need not know about any of the global vars used by expr().
         */

        register int    rval;

        Error = 0;
        Str   = expression;
        rval  = expr();
        return( (*err = Error) ?  0  :  rval );
}

/*---------------------------------------------------------------*/
```

(Continued on page 28)

 (Listing Continued, text begins on page 18)

```

static int expr()
{
        int     lval;

        lval = factor();

        switch (*Str)
        {
        case '+':  Str++;    lval += expr();      break;
        case '-':  Str++;    lval -= expr();      break;
        case '*':  Str++;    lval *= expr();      break;
        case '/':  Str++;    lval /= expr();      break;
        default :                                 break;
        }

        return( lval );
}

/*---------------------------------------------------------------------*/

static int factor()
{
        int     rval = 0 , sign = 1 ;

        if ( *Str == '-' )
        {
                sign = -1 ;
                Str++;
        }

        if ( *Str != '(' )
                rval = constant();
        else
        {
                Str++;
                rval = expr();

                if ( *Str == ')' )
                        Str++;
                else
                {
                        printf("Mis-matched parenthesis\n");
                        Error++   ;
                }
        }

        return (rval * sign);
}

/*---------------------------------------------------------------------*/

static int constant()
{
        int     rval = 0 ;

        if( !isdigit( *Str ))
                Error++;

        while ( *Str && isdigit(*Str) )
        {
                rval = (rval * 10) + (*Str - '0') ;
                Str++;
        }

        return( rval );
}
```

End Listing


The bug stops here.

by Axel Schreiner

/lib/libc.a(signal.o)

Signal(2)[1] does not *generate* a signal but is the meager protection offered against the cruelty of **kill(2)**. This column demonstrates how to deal with signals. Because this topic presupposes a broad range of knowledge, we start with a theoretical overview.

Theory

Signal is the term for a number of events that a process may encounter in a more or less unforeseen fashion:

SIGHUP
may happen when the controlling terminal is switched off (i.e., once the interface loses the carrier)

SIGINT
can happen if the interrupt key is pressed at the controlling terminal, usually break or del

SIGQUIT
can happen if the quit key is pressed at the controlling terminal, usually ^ \

SIGKILL
is signal 9, which can be sent explicitly with the **kill** command or the **kill(2)** system call

SIGSYS
results from a system call with bad parameters—primarily if programs from a foreign operating system are run under Unix

SIGPIPE
is sent to the writer of a pipe, if the corresponding reader exits

SIGALRM
is received by a process once a time interval set up by **alarm(2)** has expired

SIGTERM
is signal 15, which is sent by **kill** as a default.

There are 16 signals, ranging from 1 (**SIGHUP**) to 16. They result from events at the controlling terminal, through errors in the process itself, or through actions of other processes. Signal 16 has no predefined meaning. The list above does not contain the names of the signals provoked by arithmetic errors (e.g., division by zero), by unknown machine instructions, or by access to nonexisting memory regions; these situations are somewhat machine dependent.

The system call **kill(pid, sig)** sends signal **sig** to the process **pid**. **Kill(2)** returns a 0 on success and −1 if, for example, a signal is sent to a nonexisting process. The super-user may send signals to arbitrary recipients; everybody else can send signals only to processes with the same user number. There is a process 0, but if a signal is sent to **pid 0**, it goes to all processes in the user's own process group. The super-user, by sending a signal to **pid − 1**, reaches all processes in the entire system with the exception of processes 0 and 1. This call is really only for the benefit of */etc/init* (i.e., for the process controlling the timesharing operation).

If a process does not take defensive measures, receiving a signal is deadly. For some signals, a memory dump is produced as a file core (assuming the process has appropriate privileges with respect to the working directory, the terminated program text, etc.). **SIGINT** and **SIGQUIT** differ precisely in that only **SIGQUIT** produces a memory dump. As a rule, you can terminate a process only by using the interrupt key. If the system is short of disk space, the super-user might occasionally execute

```
find / -name core -a -exec rm {} \;
```

to locate and remove all memory-dump cores.

A process can defend itself against all signals with the exception of **SIGKILL**.

```
#include <signal.h>
    . . .
    signal(sig, SIG_IGN);
```

SIG_IGN requests that the signal **sig** not be received by the process at all.

```
int f( );
    . . .
    signal(sig, f);
```

Receipt of the signal **sig** causes the function **f** to be called, which receives the signal number as an argument.

```
#include <signal.h>
    . . .
    signal(sig, SIG_DFL);
```

This finally restores the default setting (i.e., recognition that a signal is deadly and might produce a memory dump).

Signal states (i.e., **SIG_DFL**, **SIG_IGN**, or acceptance by a function) are preserved across **fork(2)**. **SIG_IGN** or **SIG_DFL** also remain in effect across **exec(2)**. A catch function set up with **signal(2)**, of course, cannot be retained across **exec(2)** (this system call eliminates program text and data in the running process); signals connected to a function are therefore implicitly returned to **SIG_DFL** during **exec(2)**.

Once a catch function has been set up using **signal(2)**, it is invoked only once for most signals. As soon as the process receives the signal (i.e., even before the catch function is actually called, the signal setting is returned to **SIG_DFL** for the next occurrence of the signal. That results in a race condition: if a signal is received twice in rapid succession, it is very likely to be deadly.

A catch function usually has the following form:

```
#include <signal.h>

static  trap(sig)
register int sig;
{
        signal(sig, SIG_IGN);
        . . .
        signal(sig, trap);
}

main( )
{
        . . .
        signal(SIGINT, trap);
        . . .
}
```

At the beginning of the function, the signal state is set to **SIG_IGN**. At the end, the signal is reconnected to the same function. In between there can be essentially arbitrary program text; however (on the PDP-11, for example), at this point the floating-point registers have not been saved.

A catch function will typically set a global variable that is inspected in the main program at a suitable point. As an alternative, we can jump back into the main program:

```
#include <setjmp.h>
#include <signal.h>

static jmp_buf reset;

static  trap( )
{
        . . .
        longjmp(reset, 1);
}

main( )
```

```
{
        ...
        setjmp(reset);
        signal(SIGINT, trap);
        ...
}
```

Setjmp(3) arranges that a subsequent call of **longjmp( )** has a target point: the program then will return a second time from the call to **setjmp( )**. The value returned is 0 the first time and the value of the second argument to **longjmp( )** thereafter. Once the program returns to the main routine, a major command loop can be restarted and so on. This is the customary method for error and interrupt handling in programs such as **ed**.

A process can die or enter a signal-catching function only if it is ready for execution. If a process is swapped, it must be brought back into main memory before it can die! This is especially true while executing code in a device driver; a process can be blocked by the driver so that it does not become ready to execute even when it receives a signal—such processes can usually be eliminated only by a system restart. If you write device drivers, you should arrange for suitable mechanisms (**ioctl(2)**) in the event a magnetic tape transport or a floppy disk drive goes on the blink.

Take definitions in this section with a grain of salt: Berkeley Unix has more signals than, say, Unix Version 7, and there is even an additional signal state (**SIG_HOLD**) which eliminates the race condition in Unix Version 7. In an attempt to be generally applicable, our discussion is limited to Unix Version 7.

Process Group and Controlling Terminal

Starting Unix primarily means getting */etc/init* to execute as process 1. When the timesharing service is started, this process reads the file */etc/ttys* and generates one process for each terminal, which is marked appropriately in this file. Each new process is the first one in a new process group, being the first in a chain of descendants to open a connection to a terminal device. The terminal itself is the controlling terminal of the process group.

The controlling terminal is inherited during **fork(2)** and **exec(2)** even if there is no further file connection to it! The only way to obtain a process without a controlling terminal is to generate it via process 1 (in the file */etc/rc*, from which the timesharing service is started) and never to connect such a process to a terminal.

/bin/sh(wait)

Signals caused at a terminal (i.e., **SIGHUP**, **SIGINT**, and **SIGQUIT**) are always sent to all processes in a process group. An interactive shell (i.e., one connected to a terminal for standard input and output) has to defend itself against **SIGINT** and **SIGQUIT**, otherwise a terminal session would be quickly over.

Every shell ignores **SIGQUIT**. An interactive shell catches **SIGINT**, but the catch function does nothing. As a consequence, the shell command **wait** can be terminated using the interrupt key: **wait(2)** is one of the system calls terminated early by a signal.

Because **SIGINT** and **SIGQUIT** are ignored by default in a background process, one can use **wait $!** rather than the much less efficient **ps** to discover if the last background process **$!** is already done; either **wait** does not happen (i.e., the command returns immediately), or the user interrupts the wait state using **interrupt** and knows that the background process is still active.

Depending on the Bourne shell implementation, **wait** does not necessarily accept a parameter. Without a parameter, the command applies to all background processes together.

SIGTERM is also ignored by an interactive shell. This implies that you can use **kill 0** in a dire emergency to eliminate all of your own processes, because the shell itself, and thus the terminal session, is not eliminated by **SIGTERM**.

/bin/sh(logout)

The C shell has a special **logout** command. For the Bourne shell, this command can be constructed by recording the process number of the login shell using the following commands in **$HOME/.profile**:

```
SHELLID=$$; export SHELLID
```

Logout then is the following shell script (e.g., in a file */usr/bin/logout*):

```
kill -16 $SHELLID
```

With **trap "echo logout" 0** in **$HOME/.profile**, we can also report that a terminal session is about to be concluded as a response to **^D**.

/bin/nohup

A background process will ignore **SIGINT** and **SIGQUIT** but not **SIGHUP**. This can cause background processes to die as soon as the terminal is turned off following the end of a terminal session. If your terminal interface and driver operate in this fashion, you should issue such background commands with the prefix **nohup**, which will insulate the command against a **SIGHUP** signal. Also, your successor at the terminal will be happier, because standard and diagnostic output of your command will be redirected to a file **nohup.out** in your directory and not clutter up the screen.

/bin/kill

The **kill** command, Berkeley style, is used to send a signal to one or more processes. Usually, **SIGTERM** is sent, but a signal number may be specified explicitly. As an example of the **kill(2)** system call, we show a version of **kill** where the signal number may be specified mnemonically. The following Bourne shell script could be used:

```
no=
case $1 in
-[Hh][Uu][Pp])
        no=-1 shift;;
```

```
        . . .
-*)
        no=$1 shift;;
esac
kill $no $*
```

If we move */bin/kill*, for example, to */usr/bin/kill* and install this script as */bin/kill* . . . well, your system will recursively get a severe headache as soon as the next **kill** is issued. Your command search path **PATH**, which can be displayed using **echo $PATH**, normally consists of the active directory (i.e., of nothing) followed by */bin* and then */usr/bin*; **kill** in the last line of the shell script presumably will be found in */bin* (i.e., will be the shell script just issuing the command!). To avoid recursion, the last line must reference the moved original **kill** command.

By the way, a shell script always should fully qualify system commands or set **PATH** so that private commands cannot influence public shell scripts.

The C shell executes **kill** itself; the Bourne shell, however, does not. In an emergency, a user may have too many processes and no way to kill them from within the Bourne shell. The C program in Listing One (page 37) has the same capabilities as the shell script above. However, if the program is called as **exec ekill** . . . , it does not need a separate process table entry, and it will return with a new copy of the Bourne shell. Called without an argument, the program will display a list of signal names.

Observe that **exit( )** must be explicitly called at the end of **main( )**; in this program, we define **exit( )** so that either the current process calls up a Bourne shell using **exec(2)** or the process itself is terminated using **_exit( )**. By the way, **exit( )** cannot be defined local to the program (i.e., using static), otherwise library functions called by the program could still drag in the official version of **exit( )**.

We have used two functions that may be useful in other applications:

```
#include <stdio.h>

char *strsave(s)        /* save string dynamically */
register char *s;
{
        register char *save = calloc(strlen(s) + 1,
            sizeof(char));

        if (save)
                return strcpy(save, s);
        perror("strsave"), exit(1);
}
```

Although mentioned in the C book by Kernighan and Ritchie, **strsave( )** unfortunately has not made its way into the C library. This version assumes that **strcpy( )** delivers its first argument as a result—which is claimed by **string(3)** but unfortunately not by the library that **lint** on our system uses for checking library calls . . . .

The following function converts its **string(!)** argument to upper case:

```
#include <ctype.h>

char *stoupper(s)        /* string to UPPER CASE */
register char *s;
{
        register char *cp;

        for (cp = s; *cp; ++cp)
                if (islower(*cp))
                        *cp = toupper(*cp);
        return s;
}
```

Note that (at least in some implementations) **toupper( )** will deliver upper case even if the argument is not a lower-case letter! Actually, this is quite reasonable. If we already know the character to be a lower-case letter, we need not call **islower( )**; this will save only a few microseconds if the call is not made implicitly by **toupper( )**.

/etc/init

Controlling Multi-user Service

Process number 1, **init**, in some implementations of Unix Version 7 reacts to a signal and reads the file */etc/ttys* again. A terminal can be included in multi-user operations if it is mentioned in this file on a line starting with 1. If the line starts with 0, the terminal is excluded.

If **init** accepts a suitable signal, we can control multi-user service dynamically by changing */etc/ttys* appropriately and then issuing the signal. This happens in the program in Listing Two (page 38), installed as **attach** and **detach** in Perkin Elmer's Edition VII; **attach** arranges for the terminals mentioned as arguments to participate in multi-user operations and **detach** drops the terminals and eliminates the associated processes.

Use these commands with some caution; if signals are sent to **init** in rapid succession, process 1 might be eliminated and with it essentially the entire system! The program therefore must first modify the file for all arguments and then send a single signal. Because reading the file is the more expensive operation, we have elected to compare each line of the file with all arguments to the command.

The terminal name appears at the end of a line in the file. We are willing to select a terminal based on the last few letters of its name. Of course, the argument **e** will then cause console *as well as* **ttye** to be attached or detached . . . .

The program itself is quite simple: we copy */etc/ttys* into a temporary file and change the relevant lines. For efficiency, we compare each line with all arguments before writing it to the output file. Because **attach** is less destructive than **detach**, the program really must be called as **detach** for a terminal to be disconnected.

If at least one line was changed, we replace */etc/ttys* by the copy and inform **init**. Normally only the super-user can change */etc* (!), therefore only the super user can create the copy. Thus renaming should be possible:

```
#include <stdio.h>

rename(new, old)                /* rename a file */
char *new, *old;
{
        unlink(new);
        if (link(old, new) == -1)
        {
                fputs(old, stderr);
                fputs(", ", stderr);
                perror(new), exit(1);
        }
        if (unlink(old) == -1)
                perror(old), exit(1);
}
```

Blocking Signals

While */etc/ttys* is being replaced, we would prefer to be uninterrupted because we might otherwise find our system without a list of user terminals during the next bootstrap. Not even the super-user can prevent **SIGKILL**, but all other signals can be blocked with the following **disable( )** function; a subsequent call to **enable( )** will later restore the original situation:

```
#include <signal.h>

static int (*sigs[NSIG-1])( );

disable( )
{
        register int i;

        for (i = 1; i < NSIG; ++i)
                if (i != SIGKILL)
                        sigs[i-1] = signal(i, SIG_IGN);
}


enable( )
{
        register int i;

        for (i = 1; i < NSIG; ++i)
                if (i != SIGKILL)
                        signal(i, sigs[i-1]);
}
```

/lib/libc(abort.0)

At least according to chapter 3 of the manual, **abort( )** executes the PDP-11 **iot** instruction and thus normally will terminate the process with a core dump.

Beginning with Unix Version 7, one can send signals to oneself. A portable way to terminate a process with a core dump is the following:

```
#include <signal.h>
```

```
abort( )
{
        signal(SIGQUIT, SIG_DFL);
        kill(getpid( ), SIGQUIT);
}
```

/bin/stty

Depending on context, you may need to perform various cleanup tasks at various points in a program: somewhere there exists a temporary file, elsewhere a terminal echo may have been turned off, elsewhere yet a file may have the wrong protection keys, and in well-founded cases a second or third process might be on the prowl. It is relatively easy to deal with such cases, one at a time, as shown in Listing Three (page 39).

Here **echo( )** takes care of the necessary cleanup operations. If the routine was called through a signal, the same signal is subsequently sent back to the process itself.

In the general case, we should be able to stack the repairing functions by using the following functions; these mostly combine the ideas of **disable( )** and **abort( )**:

int catchall(f) int (*f)( );
declares, similar to **signal(2)**, **f** as a reaction to all signals except **SIGKILL**.

int catch(f) int (*f)( );
does the same, but only for those signals that are not ignored at present.

int recatch(i) int i;
calls the aforementioned function **f( )**, with the number of the signal effecting the call as an argument—at the beginning of this function, the signal itself has been set to **SIG_DFL**, and **recatch(i)** arranges for the signal i to be set just as it was set with the last call of **catchall( )** or **catch( )**.

int uncatch( )
arranges for all signals the setting that was in effect before the last call to **catchall( )** or **catch( )**.

The functions return 0 on success and −1 on failure.

We can use these functions to stack signal reactions. If we want to execute only the latest reaction, we conclude a trap function using **recatch( )**. If all reactions to the signal should be executed, we specify **uncatch( )** at the end of each trap function and send ourselves the signal again. At the beginning of a trap function, the signal should be set to **SIG_IGN** in either case.

The functions themselves are not very complicated. Essentially, they dynamically manage a stack on which the old settings of the signals are stored, as obtained from **signal(2)**. See Listing Four (page 40).

catch( ) contains the typical statements that should always be used if **SIGINT** or **SIGQUIT** should be caught. The first call to **signal( )** reports whether the signal is currently being ignored; in this case, the setting of the signal has not yet been changed. Only if the signal is not being ignored is it connected to the desired new reaction. These statements prevent a background process (for which these signals are ignored by default) from being subject to these signals again.

/usr/games

What does the program in Listing Five (page 40) do? Due to a complete absence of **goto** statements, this must be a so-called structured program. According to an often published opinion, however, such programs should be essentially immediately intelligible. Hint: Even if you do everything else as a super-user, don't do so here. Otherwise your system will self-destruct, and this columnist shall disavow any knowledge of this program.

Notes

[1] **Signal(2)** here denotes the function **signal( )**, explained in chapter 2 in the *Unix Programmer's Manual.*

DDJ

Unix Exchange

(Text begins on page 30)

Listing One

```
#include <stdio.h>
#include <signal.h>
#include <ctype.h>

#define eq(a,b) (strcmp((a), (b)) == 0)

static char *signals[ ] = {
    "HUP", "INT", "QUIT", "ILL", "TRAP",
    "IOT", "EMT", "FPE", "KILL", "BUS",
    "SEGV", "SYS", "PIPE", "ALRM", "TERM",
    0 };
static char eflag;      /* argv[0][0] */

exit(code)              /* terminate or new shell */
int code;
{
    if (eflag == 'e'
        && isatty(fileno(stdin))
        && isatty(fileno(stdout)))
            execl("/bin/sh", "sh", "-i", 0);
    _exit(code);
}

static int signum(name)          /* number of a signal */
char *name;
{
    register char **sig = signals,
         *ucase = stoupper(strsave(name));

    while (*sig)
        if (eq(ucase, *sig++))
        {
            cfree(ucase);
            return sig - signals;
        }
    return -1;
}
```

(Continued on next page)

Unix Exchange

Listing One

(Listing Continued, text begins on page 30)

```
main(argc, argv)
 int argc;
 register char **argv;
{
    register int sig = SIGTERM;

    eflag = **argv;
    if (*++argv && **argv == '-')
    {
            if (isdigit(*++*argv))
                sig = atoi(*argv);
            else
                sig = signum(*argv);
            if (sig <= 0 || sig >= NSIG)
            {
                fputs(*argv, stderr);
                fputs(": is not a signal\n", stderr);
                exit(1);
            }
            ++argv;
    }
    if (*argv)
            do
                if (isdigit(**argv)
                    && kill(atoi(*argv), sig) == -1)
                        perror(*argv);
            while (*++argv);
    else
            for (argv = signals; *argv; ++ argv)
                fprintf(stderr, "%d\t%s\n",
                        argv - signals + 1, *argv);
    exit(0);
}
```

End Listing One

Listing Two

```
#include <stdio.h>

#define LEN     20                          /* max. line length */
#define TTYS    "/etc/ttys"                 /* terminal table */
#define TMP     "/etc/ttys.tmpXXXXXX"       /* temporary copy */
#define TELL    kill(1, 2) == -1 && (perror("init"), exit(1))

#define eq(a,b) (strcmp((a), (b)) == 0)

main(argc, argv)
int argc;
char **argv;
{
    register char **argp;
    char buf[LEN], *last,
        *tmp = mktemp(TMP),
        set = **argv == 'd'? '0': '1';
    int change = 0;

    if (! freopen(TTYS, "r", stdin))
        perror(TTYS), exit(1);
    if (! freopen(tmp, "w", stdout))
        perror(tmp), exit(1);
    while (fgets(buf, sizeof buf, stdin))
```

```
        {
            last = buf + strlen(buf) - 1;
            if (*last != '\n')
            {
                fputs(buf, stderr);
                fputs(": too long\n", stderr);
                exit(1);
            }
            *last = '\0';
            for (argp = argv+1; *argp; ++ argp)
                if (**argp
                    && eq(last - strlen(*argp), *argp))
                        break;
            if (*argp && buf[0] != set)
            {
                ++ change;
                buf[0] = set;
            }
            puts(buf);
        }
        if (change)
        {
            fflush(stdout);
            rename(TTYS, tmp);
            TELL;
        }
        else
            unlink(tmp);
}
```

End Listing Two

Listing Three

```
#include <signal.h>
#include <sgtty.h>
#include <stdio.h>

static struct sgttyb sgttyb;

echo(i)
register int i;
{
    stty(fileno(stdout), &sgttyb);
    if (i)
        kill(getpid( ), i);
}

main( )
{
    int flags;

    gtty(fileno(stdout), &sgttyb);
    flags = sgttyb.sg_flags, sgttyb.sg_flags &= ~ECHO;
    stty(fileno(stdout), &sgttyb);
    sgttyb.sg_flags = flags;

    . . .
    signal(SIGINT, echo);
    . . .
    echo(0);
    signal(SIGINT, SIG_DFL);
    . . .
}
```

End Listing Three

(Listing Four begins on next page)

 (Listing Continued, text begins on page 30)

Listing Four

```
#include <signal.h>

static struct signals {
    int (*s_sigs[NSIG-1])( );
    struct signals *s_prev;
    } *top;

static struct signals *push( )
{
    register struct signals *p = (struct signals *)
        calloc(1, sizeof(struct signals));

    if (p)
        p→s_prev = top, top = p;
    return p;
}

int catchall(f)
register int (*f)( );
{
    register int i;
    register struct signals *p = push( );

    if (p)
    {
        for (i = 1; i < NSIG; ++ i)
                if (i != SIGKILL)
                p→s_sigs[i-1] = signal(i, f);
        return 0;
    }
    return -1;
}

int catch(f)
register int (*f)( );
{
    register int i;
    register struct signals * p = push( );

    if (p)
    {
        for (i = 1; i < NSIG; ++ i)
            if (i != SIGKILL
                && (p→s_sigs[i-1] = signal(i, SIG_IGN))
                    != SIG_IGN)
                    signal(i, f);
        return 0;
    }
    return -1;
}

int uncatch( )
{
    register int i;
    register struct signals *p = top;

    if (p)
    {
        for (i = 1; i < NSIG; ++ i)
            if (i != SIGKILL)
                signal(i, p→s_sigs[i-1]);
        top = top→s_prev;
        cfree(p);
        return 0;
    }
    return -1;
}

int recatch(i)
register int i;
{
    if (top)
    {
        signal(i, top→s_sigs[i-1]);
        return 0;
    }
    return -1;
}
```

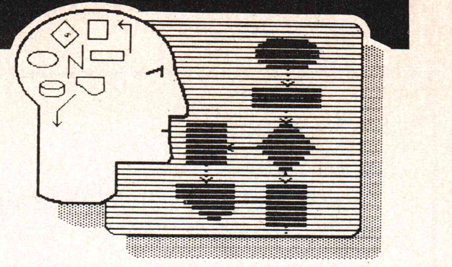
End Listing Four

Listing Five

```
#include <signal.h>

catch( ) {}

main( )
{
    int vater, ich, sohn;

    signal(16, catch);
    for (vater = 0; ich = getpid( ); vater = ich)
        switch (sohn = fork( )) {
        default:
            pause( );
        case -1:
            signal(SIGALRM, catch);
            alarm(2);
            getchar( );
            alarm(0);
            if (vater)
            {
                signal(16, catch);
                kill(vater, 16);
                pause( );
            }
            kill(sohn, 16);
            wait(0);
            exit(0);
        case 0:
            ;
        }
}
```

End Listings

THE SOFTWARE DESIGNER



by Michael Swaine

All code is built on algorithms, and all algorithms have to be communicated in code, even if only in pseudocode. So how does this month's *DDJ* more than any other month's warrant description as an algorithms issue? Well, we're presenting two articles that emphasize perennial algorithmic topics in programming: sorting and searching. Each of these articles takes a somewhat uncommon approach, and presents a program that is algorithmically distinctive. We hope they're more than distinctive, of course: we want to present algorithms that are of practical use to advanced programmers, so we've encouraged the authors to suggest applications in which their approaches might represent the algorithms of choice.

Sorting and searching algorithms are probably as thoroughly analyzed as any programming techniques. What follows here is not a definitive survey of sorting and searching techniques, but a summary of the common algorithms with the generally accepted judgements on their practical uses. None of this is new material, but it may set the stage for this month's sorting and searching algorithms.

Speed is the first performance measure we think of applying to a sort routine. But what speed? Worst case, best case, or "typical" case? And what's typical? It's known that any algorithm that sorts by comparing items must, for some sequence of **n** items, use **O(n log n)** comparisons; that is, the lower bound on the worst case is **order-(n log n)** time; but some algorithms have much worse worst-case times than this and still perform acceptably in most real-life situations. And speed isn't everything; it isn't even particularly important in some applications.

Sometimes a simple sort algorithm is the best. If you're sorting fewer than 100 items, if the sort routine will only be used once or twice and discarded, or if the data to be sorted are likely to be already nearly sorted, you probably would do just as well to implement a simple selection sort. The logic of this algorithm is lucid: find the smallest element, put it at the top of the list, and iterate down the list.

```
algorithm Selectionsort;
begin
for i:=1 to n do
   begin
   min:=i;
   for j:=i+1 to n do
      if a[j]<a[min] then min:=j;
   swap(a[i],a[min]);
   end;
end;
```

Simple algorithms are simple to debug and make it easier to convince yourself of their correctness. According to Robert Sedgewick, this simple selection sort algorithm is the technique of choice if you are sorting files with large records and small keys and actually have to perform the rearrangement, rather than just juggling indices. And it may be the technique of choice if you have to write a sort cold, have it running as quickly as possible, and then discard it. But this is an **O(n^2)** algorithm; many situations call for something more efficient.

Shellsort is a more efficient algorithm, and, although it's well-known, it is also insufficiently analyzed. Its overall efficiency has yet to be characterized. It may not be as efficient as, say, Quicksort, but it can be perfectly adequate for many applications. Its efficiency depends on the selection of a set of values that control the granularity of its early passes through the file. In the degenerate case with granularity **1** (**k = 1**), the core of the algorithm, presented here, moves gradually down the list, keeping the upper portion of the list, above the item under consideration, sorted. That's the core, and it is a sort algorithm unto itself, but the full Shellsort does more. By beginning with larger values for **k** and working down to **k = 1**, Shellsort in effect removes some of the large-scale disorder from the file, giving the **k = 1** pass better data to work with. That turns out to be a good thing, since the core algorithm, hence the **k = 1** pass, is especially sensitive to order; its worst-case performance is quadratic, its best, for a completely ordered file, linear.

Shellsort is probably the simplest algorithm that represents a common technique in sort algorithms: shifting strategies as the data becomes more organized. In the case of Shellsort the shift is accomplished by adjusting one parameter, but in some variations on Quicksort quite different algorithms are employed at different stages in the sorting.

```
algorithm Shellcore;
begin
for i:=2 to n do
   begin
   j:=i;
   while a[j-k]>a[i] do
      begin
      a[j]:=a[j-k];
      j:=j-k;
      end;
   a[j]:=a[i];
   end;
end;
```

Quicksort was invented by C. A. R. Hoare in 1960. It's been widely used,

BRIEF™

"BRIEF Is The Best Editor I Have Ever Used. I Switched. I Use BRIEF Regularly."

Steve McMahon*

Choosing the right program editor is an extremely important decision. After all, more time is spent with your program editor than with any other category of software. A good program editor helps to stimulate creativity and produce programs of superior design. A poor program editor, on the other hand, can be worse than none at all. It can slow you down, get in your way, and make even the simplest tasks difficult.

BRIEF - PROGRAM EDITING *YOUR WAY*

Every programmer has an individual style that makes their work unique. Most program editors compromise that style by forcing the programmer to conform to their methods. Not so with BRIEF.

BRIEF's most powerful feature is its ability to conform to your way of programming. BRIEF can be easily tailored/customized to your individual preferences. For example, keyboard reassignment allows the keyboard to be used in whatever way you prefer. Keystroke macros allow long sequences of editor directives to be repeated by pressing a single key. Plus you can simultaneously work with numerous program and data files, and configure many windows to control BRIEF's visual presentation.

75% OF THE EXPERTS WHO HAVE TRIED BRIEF SWITCHED!

(Call For Details)

"A BONA FIDE UNDO" Steve McMahon – BYTE REVIEW "BUILD YOUR DREAM EDITOR" MARCH 1985

Here are Steve McMahon's exact words: "...BRIEF implements a true undo facility, by default allowing command-by-command recovery from the last 30 commands...The number of commands you want to be able to undo can be changed." (up to 300) "Only with BRIEF, though, was it possible to undo a macro that produced 4000 words of text with a single keystroke."

EVERY FEATURE YOU WANT

BRIEF supports practically every feature you've ever seen, and some you probably haven't thought of yet.

- Edit Multiple Large Files
- Windows (Tiled & Pop-up)
- Unlimited File Size
- Full UNDO (N Times)
- True Automatic Indent for C
- Exit to DOS Inside BRIEF
- Compile programs inside BRIEF
- Uses All Available Memory
- Intuitive Commands
- Tutorial
- Repeat Keystroke Sequences
- 15 Minute Learning
- Reconfigurable Keyboard
- Online Help
- Search for "Regular Expressions"
- Mnemonic Key Assignments
- Horizontal Scrolling
- Comprehensive Error Recovery
- And ... a Complete, Compiled, Programmable and Readable Macro Language.

```
======================== make.c ========================
int       handle = 0;

main (argc, argv█

int              argc;
========= fsa.h ==========|============== makefile.h =========
#include "..\include\ctype.h"  |/**
                               | **            makefile.h:
typedef  struct                | **
{                              | **       This is the definitions fil
   short     action,           | **   Hopefully, it won't be unreasonab
             state;            | **   that have been written.
} Fsa;                         | **/
#ifdef   FSA_MAIN              |
Fsa      fsa[][8] = {/* Alphanum    Co|typedef struct cmd_struct
                                      |{
/* State 0. */          0, 2,      10 |   char          *cmd_text;
/* State 1. */          10, 0,     10 |
/* State 2. */          0, 2,      1, |   struct cmd_struct *next_cmd;
/* State 3. */          0, 5,      11 |} *Cmd_Ptr, Cmd;
/* State 4. */          0, 4,      0, |
Mismatched open parenthesis.                Line: 11    Col: 17       2:17 pm
```

A TYPICAL BRIEF SCREEN

Notice there are three windows on the screen simultaneously and each one is viewing a different file. The mainline of a C program is visible in the uppermost window; the programmer has run a syntax checking macro which found a mismatched open parenthesis in the arguments to the mainline. The other two windows show header files containing information crucial to the design of the program. BRIEF can have an unlimited number of windows and files accessed simultaneously.

BRIEF'S ONLY LIMITATIONS ARE THE ONES *YOU* SET

Full refund if not satisfied in 30 days. ONLY $195

→ **FREE** with order: "Best of BRIEF Macros" — Contest entries include macros for Fortran, C, Calculator, Base Converter, etc. Call before November 30. **800-821-2492**

AVAILABILITY: IBM-PC and Compatibles, AT, & TANDY 2000

Solution Systems™

335-D Washington St., Norwell, MA 02061 (617) 659-1571 **800-821-2492.**

Steve McMahon's quote courtesy Suntype Publishing Systems. BYTE review by Mr. McMahon may be found in BYTE Magazine March 1985.

Circle **no. 102** on reader service card.

deeply analyzed, and much tweaked since then. It has a worst-case **O(n^2)** performance, but has **O(n log n)** expected time under the assumption that all permutations of the to-be-sorted elements are equally likely. Quicksort is fast and relatively memory efficient, and is often used in library routines.

Quicksort is defined here as a recursive procedure. Its arguments, **left** and **right**, define the endpoints of the sort. **Middle** is a point selected by procedure **Partition**. **Partition** is any procedure that puts the middle-th element (or identical elements) in its (their) proper place and ensures that all elements to the left of it (them) are less than it (them), and all elements to its (their) right are greater. **M** is the number of identical elements that **Partition** puts in their proper place.

```
algorithm Quicksort(left,right);
begin
if right>left then
   begin
   Partition(left,middle,right);
   Quicksort(left, middle - 1);
   Quicksort(middle+m,right);
   end;
end;
```

Various methods have been employed to improve Quicksort's performance, such as R. S. Scowen's Quickersort and M. H. van Emden's Qsort (algorithms 271 and 402 in the *Collected Algorithms of the ACM*, respectively). Unwinding the recursion is one technique that can increase the speed. Another method involves abandoning the algorithm for something like Shellcore for very small subfiles. A third technique helps to avoid worst-case times: it involves using some sort of randomizing technique in **Partition** to select the middle point.

Heapsort is an efficient sort algorithm that uses a data structure that has applications in queuing. One of its virtues is that it has **O(n log n)** as a worst-case time. The algorithm works by constructing a semiordered tree structure called a heap, sawing off the root (which will be the largest element remaining), rebuilding the resulting forest into another heap, and iterating. Procedure **Heapprop** rearranges items to ensure that the heap property applies. **Makeheap** constructs the initial heap.

```
algorithm Heapsort;
begin
Makeheap;
for i:=n to 2 step -1 do
   begin
   swap(a[1],a[i]);
   Heapprop(i-1);
   end;
end;
```

Bubblesort is a curious phenomenon. I learned it early in my programming experience. You probably did, too. Does anyone know why? It is neither very simple nor very efficient, and its detailed operation is less clear than that of other sorts, although its operation is not hard to describe. Essentially, it consists of a number of passes through the file, swapping adjacent elements as necessary.

```
algorithm Bubblesort;
begin
repeat
   t:=a[1];
   for j:=2 to n do
      if a[j-1]>a[j] then
         begin
            t:=a[j-1];
            swap(a[j],a[j-1]);
         end;
until t=a[1];
end;
```

How you sort depends on what you have to sort, including how it is already arranged as well as other aspects of the data. Another criterion you might apply to a sort algorithm is stability. If you intend to sort on more than one key and not have the sorting on one key destroy the sorting on the other, you need a stable sort. There are instances in which this is not necessary, and there is much power in knowing what you can get along without. In this issue (page 68) we present a sort algorithm that has some speed advantage when stability is not a consideration.

Knowing that the data is not randomly arranged can also have a marked effect on the expected performance of the algorithms. One of the most common sorting situations involves adding a few items to an already sorted file; here the more sophisticated algorithms can't compete with such simple algorithms as Shellcore. Finally, in sorting files too large for in-core methods, tape or disk access speed limitations can override all other considerations in searching for a fast algorithm. The basic algorithm for external sorts is: sort (relatively) small chunks of the file in memory, then merge these sorted chunks. Much of the analysis of external sort algorithms has to do with minimizing trips to the external well. In a virtual memory system, Sedgewick argues, it is reasonable for certain in-core algorithms (like Quicksort) to ignore the problem of external reaches entirely, since the sort's minimization criteria coincide with those of the designer of the virtual memory system.

Searching algorithms encompass a large chunk of computer science. Parsing, for example, is built on pattern matching, which is searching. Efficient implementation of a database hinges on good search algorithms. Move-search routines in chess programs implement some very complex strategies for finding things. Search techniques depend intimately on the data structure being searched, and many search algorithms are fundamentally more complex than the sort algorithms presented here. Search algorithms are harder to specify in a few lines.

The simplest search algorithm is a sequential scan of the file, and it has an advantage in addition to its simplicity: it does not upset the organization of the file. Generally, though, speed is a consideration, and an algorithm like a binary search is warranted. Binary search splits the (sorted) file more or less in half at each step, restricting its subsequent search to the appropriate half of the remaining file. This cuts search time from a linear function of the number of items in the file to a logarithmic function. Binary search can proceed implicitly on a linearly-organized file or can involve explicit construction of a binary tree. Other search methods involve

using B-trees and hash functions, two subjects we expect to expand on in future issues.

This issue of *DDJ* focuses on useful algorithms, and is intended to serve as a prelude to a new emphasis on algorithmic analysis of the program listings in *DDJ*. The reason for such an emphasis is simple: to increase the portability and usability of the tools we publish. Processor-specific magazines can get away with publishing processor-specific code. We don't want, and we think you don't want *DDJ* to become a processor-specific magazine, so we're trying to broaden the applicability of our offerings by providing the keys to allow you to extract what you can use from the listings. Those keys include, we think, well-documented high level code and clear discussion of the underlying algorithms and useful techniques embodied in the code.

And what about someday-useful algorithms? The algorithmic cutting edge? The Fgrep algorithm in this issue uses limited parallelism to gain speed. Next month we'll look at the Novix Forth chip and what it gains from a little parallelism. But deep parallelism of the sort realizable in multiprocessor machines of the future will require different algorithms, different models of the computer. What lies beyond Von Neumann architecture? We hope to investigate parallelism in upcoming issues of *DDJ*.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 194**.

Parallel Pattern Matching and Fgrep

by Ian Ashdown

The file-search utility Fgrep is not as flexible as Grep, but its parallel pattern-matching algorithms allow for one-pass processing.

Preparing an index to a large technical book; finding all references to one person in several years' worth of minutes of meetings; searching for all occurrences of a specific sequence of events in the data obtained from a scientific experiment: all of these problems and many more are examples of searching for patterns in a set of data. The question is: What is the most efficient search algorithm to use?

For single patterns, such as searching for the phrase "binary tree" in a text file, two algorithms are of particular interest: the Boyer-Moore algorithm[1] and the Knuth-Morris-Pratt algorithm.[2] Both are fast and reasonably simple to implement. However, neither is appropriate when more than one pattern at a time must be searched for.

An algorithm that is appropriate for multiple patterns appeared in a paper published in the June 1975 issue of *Communications of the ACM*.[3] Entitled "Efficient String Matching: An Aid to Bibliographic Search" and written by Alfred Aho and Margaret Corasick of Bell Laboratories, the paper presents "a simple and efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text." Without modification to the algorithm, "keyword" can be taken to mean any pattern and "a string of text" to mean any sequence of symbols, be it ASCII text, digitized data obtained from a video camera, or whatever.

The algorithm has some interesting features. For instance, most pattern-matching schemes must employ some form of backtracking, or rescanning of the string, when an attempted match to a pattern fails or multiple patterns are to be matched. The Aho-Corasick algorithm differs in that it searches for all of the patterns in parallel. By doing so, it can process the string in one pass.

The algorithm also recognizes patterns that overlap in the string. For example, if the patterns are "he," "she," and "hers," the algorithm will correctly identify matches to all three in the string "ushers."

As for speed of execution, it is independent of the number of patterns to be matched! This surprising feature is a direct result of searching for the patterns in parallel.

Curiously, this algorithm has all but disappeared from the literature of computer science. Of the hundreds of textbooks and articles written since that time that discuss pattern matching, only a few refer to the paper, and none that I am aware of present the Aho-Corasick algorithm itself.

On the other hand, if you work with Unix you may have used a utility based on the algorithm: Fgrep. This useful tool searches for and displays all occurrences of a set of keywords and phrases in one or more text files. Although it does not accept wildcard characters in its patterns like Unix's more flexible grep utility, fgrep does illustrate the speed of the Aho-Corasick algorithm. Typically, fgrep is five to ten times faster than grep in searching files for fixed string patterns.

Ian Ashdown, byHeart Software, 1089 W. 21st St., North Vancouver, British Columbia V7P 2C6 Canada

Given the algorithm's general usefulness, I thought it appropriate to reintroduce Aho and Corasick's work in this article. At the same time, it seemed a shame that fgrep has been restricted to Unix. Therefore, the source code in C for a full implementation of fgrep[4] has been included (see the listing, page 54). This serves not only to demonstrate the algorithm but also to bring the idea of a public domain Unix one small step closer to reality.

Inside the Machine

The Aho-Corasick algorithm consists of two phases: building a finite state automaton (FSA) then running a string of data through it, with each consecutive symbol considered a separate input. Before we analyze these phases, a quick summary of what finite state automata are and how they work is in order. (For a more detailed discussion, see Aho and Ullman's book on compiler design.[5])

In essence, an FSA is a conceptual machine that can be in any one of a finite number of "states." It also has a set of "state transition" rules and a set of "inputs," which together with the set of states define what inputs cause the machine to change from one state to another. Finally, because a machine serves no purpose without producing some output, an FSA has one or more "terminal" states. Output is produced only when the FSA enters such states.

A physical example of an FSA could be a light switch. This device has two states, on and off. Its inputs consist of someone pushing the switch up or down. On is a terminal state, where the associated lamp produces visible radiation. The state transition rules can be presented in a simple truth table:

	Off	On
Up	On	–
Down	–	Off

Alternatively, the rules could be shown as a state transition diagram (Figure 1, page 47) where no state transition on a particular input (as shown in the truth table) is

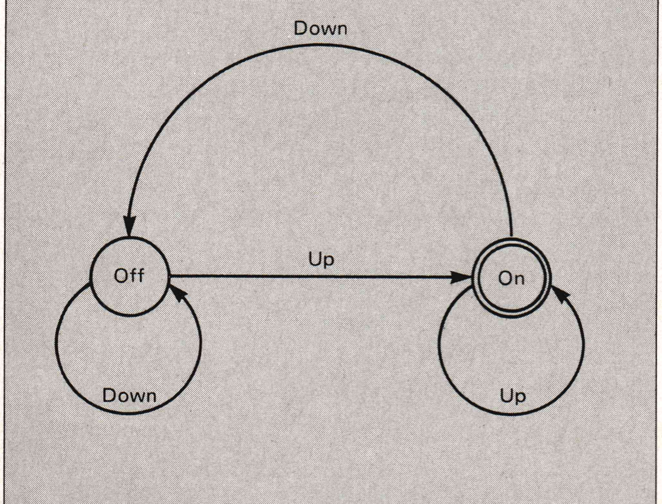


Figure 1

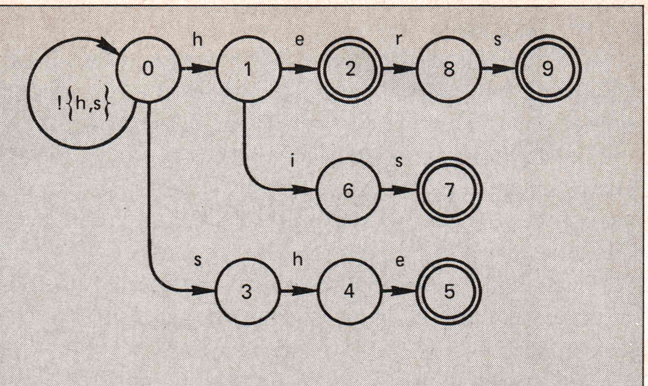


Figure 2a
Goto function

Current State:	1	2	3	4	5	6	7	8	9
Failure State:	0	0	0	1	2	0	3	0	3

Figure 2b
Failure function

Current State:	Output:
0	--
1	--
2	{he}
3	--
4	--
5	{she, he}
6	--
7	{his}
8	--
9	{hers}

Figure 2c
Output function

Current State:	Input Symbol:	Next State:	Current State:	Input Symbol:	Next State:
0	h	1	4	e	5
	s	3		i	6
	.	0		h	1
1	e	2		s	3
	i	6		.	0
	h	1	6	s	7
	s	3		h	1
	.	0		.	0
2,5	r	8	8	s	9
	h	1		h	1
	s	3		.	0
	.	0			
3,7,9	h	4			
	s	3			
	.	0			

where "." represents any symbol not included in the list of symbols for the indicated state(s).

Figure 2d
Move function

equivalent to a transition from a state to itself (as shown in the diagram).

Getting ahead of ourselves for a moment, let's look at Figure 2 (page 47). Figure 2a shows part of an FSA (the other parts shown in Figures 2b, 2c, and 2d, will be explained shortly). By having sequences of states, the FSA can recognize patterns in an input stream. For example, presenting the string "she" to the FSA shown in Figure 2a would cause it to change from State 0 to States 3, 4, and 5 as each input symbol of the string is processed. State 5 is a terminal state that indicates that the pattern "she" was recognized.

Two types of FSA can be built, one nondeterministic (Algorithm 1, page 48) and the other deterministic (Algorithm 2, page 48). The nondeterministic one (NFSA) uses functions called go_to, failure and output, while the deterministic FSA (DFSA) uses move and output. The difference between the two is that the NFSA can make one or more state transitions per input symbol, but the DFSA makes only one. With fewer transitions to make, a DFSA can process its input in less time than an equivalent NFSA.

The go_to function accepts as input parameters the current state of the NFSA and the current input symbol, and it returns either a state number (the go_to state transition), if the input symbol matches that expected by the patterns being matched, or else a unique symbol called FAIL. The only exception to this is State 0: go_to(0,X) returns a state number for all input symbols X. If symbol X does not match the beginning symbol of any of the patterns, the state number returned is 0.

The failure function is called whenever the go_to function returns FAIL. Accepting the current state as its input parameter, it always returns a state number, the failure state transition.

For the DFSA, the move function accepts the current state number and input symbol and always returns the next state number. There are no failure state transitions in deterministic FSAs.

Both the NFSA and the DFSA use the output function. As defined by Aho and Corasick, this function prints the patterns as they are matched by the FSA. However, the definition of this function can be much more general. In fact, the FSA can be implemented in such a way that it executes any arbitrary set of procedures whenever it recognizes a pattern.

FSAs used by the Aho-Corasick algorithm can be either nondeterministic or deterministic. Although a DFSA executes more quickly than its equivalent NFSA, it also requires more memory to encode its state transition tables. Which type you choose for the algorithm depends upon the application and the constraints of execution speed and memory usage.

As an example, assume a set of patterns to be matched, "he," "she," "his," and "hers," with ASCII characters as the set of input symbols. (These have been purloined from Aho and Corasick.) The FSAs needed to recognize these patterns are shown in Figure 2. Let's look at the NFSA first. Taking as input the string "ushers," the machine is run using Algorithm 1. Starting in State 0, the first symbol from the string is "u." Because only symbols "h" and "s" lead from State 0 to other states, go_to(0,u) returns 0, and the NFSA remains in State 0.

The next symbol from the string is "s." As shown in Figure 2a, the NFSA makes a go_to state transition to State 3. The next symbol ("h") causes a go_to transition to State 4, and the next ("e") to State 5. An output is defined for State 5 (Figure 2c), and the NFSA, having recognized two pattern matches, prints "she,he."

The next character is "r." No go_to transitions are defined for State 5, so go_to(5,r) returns FAIL. This causes failure(5) to return 2 (from Figure 2b), making the NFSA perform a failure transition to State 2. From here, Algorithm 1 executes go_to(2,r), which causes the NFSA to enter State 8.

Finally, the last input symbol, "s," leads the NFSA to enter terminal State 9, and the output defined for this state causes "hers" to be printed. In all, Algorithm 1 recognized the patterns "he," "she," and "hers" in the string "ushers." The state transitions made can be summarized as:

Input Symbol:	u	s	h	e		r	s	
Current State:	0	0	3	4	5	2	8	9

```
Input:   A string of symbols and a pattern-matching machine
         consisting of functions go_to, failure and output.
Output:  Matched patterns.

begin
  state = 0
  while there are more symbols in the string do
    begin
      a = next symbol in the string
      while go_to(state,a) == FAIL do
        state = failure(state)
      state = go_to(state,a)
      if output(state) != NULL then
        print output(state)
    end
end
```

Algorithm 1
Nondeterministic Pattern-Matching Machine

```
Input:   A string of symbols and a pattern-matching machine
         consisting of functions move and output.
Output:  Matched patterns.

begin
  state = 0
  while there are more symbols in the string do
    begin
      a = next symbol in the string
      state = move(state,a)
      if output(state) != NULL then
        print output(state)
    end
end
```

Algorithm 2
Deterministic Pattern-Matching Machine

Now let's look at the DFSA. Using the same input string, "ushers," this machine makes move state transitions in accordance with Algorithm 2 and Figure 2d. Its state transitions can be summarized as:

Input Symbol:	u s h e r s
Current State:	0 0 3 4 5 8 9

The only difference is that the failure transition to State 2 was skipped. By avoiding any failure transitions, Algorithm 2 can recognize a pattern in fewer state transitions and hence less time than Algorithm 1.

Software Construction

Having seen how FSAs work, we will now look at how they are built, starting with the computation of the go_to function by Algorithm 3 (page 49).

The go_to function is initially defined to return FAIL for every input symbol and every state. Each pattern is run through procedure "enter," which tries to match the pattern, symbol by symbol to the existing partially constructed go_to function. When a failure occurs, a new state is created and added to the go_to function, using the current symbol of the pattern as the input for the state

```
Input:    Array of patterns {y[1],y[2] . . . . y[k]} where each
          pattern is a string (one-dimensional array) of symbols.
          (Function go_to(s,a) is defined to return FAIL for all
          states "s" and all input symbols "a" until defined
          otherwise by procedure enter.)
Output:   Function go_to and partially computed function output.

begin
  newstate = 0
  for i = 1 to i = k do
    enter(y[i])
  for each symbol a do
    if go_to(0,a) == FAIL then
      go_to(0,a) = 0
end

procedure enter(pattern)          /* "pattern" is an array of */
begin                             /* "m" symbols */
  state = 0
  j = 1
  while go_to(state,pattern[j]) != FAIL do
    begin
      state = go_to(state,pattern[j])
      j = j + 1
    end
  for p = j to p = m do
    begin
      newstate = newstate + 1
      output(newstate) = NULL
      go_to(state,pattern[p]) = newstate
      state = newstate
    end
  output(state) = pattern
end
```

Algorithm 3
Computation of go_to Transitions

transition. Thereafter, each consecutive symbol of the pattern causes a new state to be created and added.

When the last symbol of each pattern has been processed by procedure "enter," its associated state is made a terminal state. As shown in Algorithm 3, this means that the output for the state is defined as the current pattern, which Algorithm 1 will print when the NFSA is run. However, it is easy to see that you can rewrite the statement "output(state) = pattern" in Algorithm 3 to assign whatever procedures to output(state) you want.

Finally, after all of the patterns have been processed, go_to(0,X) is redefined to return 0 for all symbols X that still return FAIL.

When Algorithm 3 completes, it has computed the go_to function of the FSA and partially computed the output function. If you simulate Algorithm 3 by hand with "he," "she," "his," and "hers" as its patterns, you will see that the output for State 5 will be "she," not "she,he." It remains for Algorithm 4 (page 51) to complete the output function as it computes the failure function.

Let's define the depth of a state as the number of go_to state transitions that must be made from State 0 to reach it. For example, the depth of State 4 in Figure 2a is 2, while that of State 9 is 4. The failure state transition for all states of depth 1 is State 0. The algorithm used to compute the failure transitions for all states of depth greater than 1 can be expressed in its simplest form as:

```
for each depth D do
  for each state S of depth D-1 do
    for each input symbol A do
      if (T = go_to(S,A)) != FAIL then
        begin
          R = failure(S)
          while go_to(R,A) == FAIL do
            R = failure(R)
          failure(T) = go_to(R,A)
        end
```

To demonstrate this using our example, let's compute one failure transition for a state of depth 2. The states of depth 1 are 1 and 3. The only input symbols for which the go_to function does not return FAIL are "e" and "i" for State 1 and "h" for State 3. Taking State 3 for "S" and symbol "h" for "A" above, we have "T" being 4, "R" being "failure(3)", which is 0, go_to(0,h) returning 1, and finally "failure(4)" being set to go_to(0,h). The failure transition for State 4 is thus State 1.

Algorithm 4 expands on the above algorithm in two ways. First, it uses a queue (a first-in, first-out list) to maintain the order of states by depth to be processed. Second, it accepts as input the output function and combines the outputs of the terminal states where appropriate. In our example, the output "he" of State 2 would be combined with the output "she" of State 5 to produce the final and correct "she,he" output for State 5.

The "move" function is computed from the go_to and failure functions by means of Algorithm 5 (page 51). Essentially, all this algorithm does is precompute all possible sequences of failure state transitions. It is similar to Algo-

```
Input:    Functions go_to and output from Algorithm 3.
Output:   Functions failure and output.

begin
  initialize queue
  for each symbol a do
    if (r = go_to(0,a)) != 0 then
      begin
        add r to tail of queue
        failure(r) = 0
      end
  while queue is not empty do
    begin
      s = head of queue
      remove head of queue
      for each symbol a do
        if (t = go_to(s,a)) != FAIL then
          begin
            add t to tail of queue
            r = failure(s)
            while go_to(r,a) == FAIL do
              r = failure(r)
            failure(t) = go_to(r,a)
            output(t) = output(t) + output(failure(t))
          end
    end
end
```

Algorithm 4
Computation of Failure Transitions

```
Input:    Function go_to from Algorithm 3 and function failure
          from Algorithm 4.
Output:   Function move.

begin
  initialize queue
  for each symbol a do
    begin
      move(0,a) = go_to(0,a)
      if (r = go_to(0,a)) != 0 then
        add r to tail of queue
    end
  while queue is not empty do
    begin
      s = head of queue
      remove head of queue
      for each symbol a do
        if (t = go_to(s,a)) != FAIL then
          begin
            add t to tail of queue
            move(s,a) = t
          end
        else
          move(s,a) = move(failure(s),a)
    end
end
```

Algorithm 5
Computation of Move Transitions

rithm 4. Because there is considerable redundancy between them, they can be merged to compute the failure and move functions concurrently, as is shown in Algorithm 6 (page 53).

Details, Details

So far, the functions go_to, failure, move, and output have been shown as something that you assign outputs to for specified inputs. This assumes a much higher-level programming language than most of today's offerings. Implementing these algorithms in C, Pascal, BASIC, or Fortran requires some extra code and work.

Aho and Corasick programmed their original version in Fortran. This is evident from their paper, where they suggest that the failure and output functions be implemented as one-dimensional arrays accessed by state numbers. With a language such as C that supports complex data structures, the code for the functions can be more elegantly written by replacing the state numbers and arrays with dynamically allocated structures. Each structure can have as members pointers to linked lists of go_to and move transitions, a pointer to the appropriate failure transition, and a pointer to a character string for the output function.

Aho and Corasick also note that the go_to and move functions could be implemented as two-dimensional arrays, with the number of states forming one dimension and the set of input symbols forming the other. However, this would require enormous amounts of memory for an ASCII character set and several hundred states. Their recommendation was that the go_to and move transitions for each state be implemented as linked linear lists or binary trees.

Because the FSA will usually spend most of its time in State 0 for large input symbol sets, it is advantageous to have the go_to or move functions execute as quickly as possible for this state. Therefore, as Aho and Corasick suggest, a one-dimensional array can be assigned to State 0. (Note that because no failure transitions are defined for State 0, its go_to and move transitions are one and the same.) The array is accessed directly, using the input symbol as the index, with the corresponding array entry being the state transition for that symbol.

An improvement not covered by Aho and Corasick can be made when you realize that several states often will have the same move transitions (see Figure 2d as an example). In such cases, the state structures can be assigned a common pointer to one linked list of move transitions.

Some applications may call for the algorithm to be coded with predefined patterns in read-only memory Here it is usually desirable to maximize the speed of execution while minimizing the code size. Aho and Ullman discuss a method of encoding the state transition tables that combines the compactness of linked lists with the speed of di-

rect array access. Their method (which Unix's yacc compiler-compiler utility uses to encode its parsing tables) is unfortunately too involved to discuss here. Interested readers are referred to Aho and Ullman's book for details.

Putting It All To Work

Implementing the Aho-Corasick algorithm is fairly straightforward, although, as you can see, the code required to flesh it out to a full emulation of Unix's fgrep is somewhat involved. If you want to use the algorithm in other programs, simply remove the fgrep shell and add whatever interface routines you require.

As an information retrieval utility that searches arbitrary text files, fgrep is useful but by no means complete. One helpful extension would be the ability to specify Boolean operators (AND, OR, and XOR) for combinations of patterns.

Also, rather than simply print matched patterns for its output, the FSA can execute whatever procedures you choose to associate with the patterns as they are matched. From this you could create a macro processor, where the FSA accepts an input string, finds matches to predefined patterns, substitutes the corresponding macro expansions, and emits the result as an output string.

For those with RAM memory to spare (a megabyte or so), consider how fast a spelling checker that makes no disk accesses beyond initially loading itself could be made to run . . . fifty thousand or more words in RAM and your text file is checked almost as fast as it can be read.

If you come up with an original and fascinating use for the Aho-Corasick algorithm, or if you derive new utilities from fgrep, by all means let me know about it. Better yet, donate your code to a public domain software group. That alone would repay me for developing the code and writing this article.

References

[1] Boyer, R. S., and J. S. Moore, "A Fast String Searching Algorithm," *CACM* 20:10 (October 1977), pp. 762 – 772.

[2] Knuth, D. E., J. H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Comp., 6:2* (June 1977), pp. 323–350.

[3] Aho, A. V., and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *CACM* 18:6 (June 1975), pp. 333–340.

[4] Bell Telephone Laboratories, *Unix Programmer's Manual*, Volume 1, Holt, Rinehart and Winston, 1983, pp. 70–71.

[5] Aho, A. V., and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, 1977, pp. 73–124.

[6] *See also* Aoe, J., Y. Yamamoto, and R. Shimada, "A Method for Improving String Pattern Matching Machines," *IEEE Transactions On Software Engineering, SE-10:1 (January 1984), pp. 116*–120.

DDJ

(Listing begins on next page)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195.**

```
Input:    Functions go_to and output from Algorithm 3.
Output:   Function move.
begin
  initialize queue
  for each symbol a do
    begin
      move(0,a) = go_to(0,a)
      if (r = go_to(0,a)) != 0 then
        begin
          add r to tail of queue
          failure(r) = 0
        end
    end
  while queue is not empty do
    begin
      s = head of queue
      remove head of queue
      for each symbol a do
        if (t = go_to(s,a)) != FAIL then
          begin
            add t to tail of queue
            r = failure(s)
            while go_to(r,a) == FAIL do
              r = failure(r)
            failure(t) = go_to(r,a)
            output(t) = output(t) + output(failure(t))
            move(s,a) = t
          end
        else
          move(s,a) = move(failure(s),a)
    end
end
```

Algorithm 6
Merged Computation of Failure and Move Transitions

Fgrep Listing

(Text begins on page 46)

```
/* FGREP.C - Search File(s) For Fixed Pattern(s)
 *
 * Version 1.03        February 11th, 1985
 *
 * Modifications:
 *
 *   V1.00 (84/12/01)    - beta test release
 *   V1.01 (85/01/01)    - added -P option
 *                       - improved command line validation
 *   V1.02 (85/01/06)    - modified "run_fsa()" and "bd_move()"
 *   V1.03 (85/02/11)    - added -S option
 *
 * Copyright 1985:     Ian Ashdown
 *                     byHeart Software
 *                     1089 West 21st Street
 *                     North Vancouver, B.C. V7P 2C6
 *                     Canada
 *
 * This program may be copied for personal, non-commercial use
 * only, provided that the above copyright notice is included in
 * all copies of the source code. Copying for any other use
 * without previously obtaining the written permission of the
 * author is prohibited.
 *
 * Machine readable versions of this program may be purchased for
 * $35.00 (U.S.) from byHeart Software. Supported disk formats
 * are CP/M 8" SSSD and PC-DOS (v2.x) 5-1/4" DSDD.
 *
 * Notes:
 *
 * The algorithm used in this program constructs a deterministic
 * finite state automaton (FSA) for pattern matching from the sub-
 * strings, then uses the FSA to process the text string in one
 * pass. The time taken to construct the FSA is proportional to
 * the sum of the lengths of the the substrings. The number of
 * state transitions made by the FSA in processing the text
 * string is independent of the number of substrings.
 *
 * Algorithm Source:
 *
 * "Efficient String Matching: An Aid to Bibliographic Search"
 * Alfred V. Aho & Margaret J. Corasick
 * Communications of the ACM
 * pp. 333 - 340, Vol. 18 No. 6 (June '75)
 *
 * USAGE: fgrep [-vclnhyefxps] [strings] <files>
 *
 * where:
 *
 *         -v      All lines but those matching are printed.
 *         -c      Only a count of the matching lines is printed.
 *         -l      The names of the files with matching lines are
 *                 listed (once), separated by newlines.
 *         -n      Each line is preceded by its line number in the
 *                 file.
 *         -h      Do not print filename headers with output lines.
 *         -y      All characters in the file are mapped to upper
 *                 case before matching. (This is the default if the
 *                 string is given in the command line under CP/M,
 *                 as CP/M maps everything on the command line to
 *                 upper case. Use the -f option if you need both
 *                 lower and upper case.) Not a true UNIX "fgrep"
 *                 option (normally available under "grep" only),
 *                 but too useful to leave out.
 *         -e      <string>. Same as a string argument, but useful
 *                 when the string begins with a '-'.
 *         -f      <file>. The strings (separated by newlines) are
 *                 taken from a file. If several strings are listed
 *                 in the file, then a match is flagged if any of
 *                 the strings are matched. If -f is given, any
 *                 following argument on the command line is taken
 *                 to be a filename.
 *         -x      Only lines matched in their entirety are printed.
 *         -p      Each matched line is preceded by the matching
 *                 substring(s). Not a UNIX "fgrep" option, but too
 *                 useful to leave out.
 *         -s      No output is produced, only status. Used when
 *                 when "fgrep" is run as a process that returns a
 *                 status value to its parent process. Under CP/M, a
```

(Continued on page 56)

Circle **no. 97** on reader service card.

Fgrep Listing

(Listing Continued, text begins on page 46)

```
 *          non-zero value returned by "exit()" may terminate
 *          a submit file that initiated the program,
 *          although this is implementation-dependent.
 *
 * DIAGNOSTICS:
 *
 * Exit status is 0 if any matches are found, 1 if none, 2 for
 * error condition.
 *
 * BUGS:
 *
 * The following UNIX-specific option is not supported:
 *
 *      -b      Each line is preceded by the block number in
 *              which it was found
 *
 * Lines are limited to 256 characters.
 *
 */

/*** Definitions ***/

#define TRUE            -1
#define FALSE            0
#define MAX_LINE       257   /* Maximum number of characters */
                             /* per line plus NULL delimiter */
#define CP/M                 /* Comment out for compilation */
                             /* under UNIX */

#define CMD_ERR    0    /* Error codes */
#define OPT_ERR    1
#define INP_ERR    2
#define STR_ERR    3
#define MEM_ERR    4

/*** Typedefs ***/

typedef int BOOL;       /* Boolean flag */

/*** Data Structures ***/

/* Queue element */

typedef struct queue
{
  struct state_el *st_ptr;
  struct queue *next_el;
} QUEUE;

/* Transition element */

typedef struct transition
{
  char lchar;                   /* Transition character */
  struct state_el *nextst_ptr;  /* Transition state pointer */
  struct transition *next_el;
} TRANSITION;

/* FSA state element */

typedef struct state_el
{
  TRANSITION *go_ls;     /* Pointer to head of "go" list */
  TRANSITION *mv_ls;     /* Pointer to head of "move" list */
  struct state_el *fail_state;  /* "failure" transition state */
  char *out_str;         /* Terminal state message (if any) */
} FSA;

/*** Global Variables and Structures ***/

/* Dummy "failure" state */

FSA FAIL_STATE;

/* Define a separate data structure for State 0 of the FSA to
 * speed processing of the input while the FSA is in that state.
 * Since the Aho-Corasick algorithm only defines "go" transitions
 * for this state (one for each valid input character) and no
 * "failure" transitions or output messages, only an array of
```

```
 * "go" transition state numbers is needed. The array is accessed
 * directly, using the input character as the index.
 */

FSA *MZ[128];   /* State 0 of FSA */

BOOL vflag = FALSE,      /* Command-line option flags */
     cflag = FALSE,
     lflag = FALSE,
     nflag = FALSE,
     hflag = FALSE,
     yflag = FALSE,
     eflag = FALSE,
     fflag = FALSE,
     xflag = FALSE,
     pflag = FALSE,
     sflag = FALSE;

/*** Include Files ***/

#include <stdio.h>
#include <ctype.h>

/*** Main Body of Program ***/

int main(argc,argv)
int argc;
char **argv;
{
  char *temp;
  BOOL match_flag = FALSE,
       proc_file();
  void bd_go(),
       bd_move(),
       error();

  /* Check for minimum number of command line arguments */

  if(argc < 2)
    error(CMD_ERR,NULL);

  /* Parse the command line for user-selected options */


  while(--argc && (*++argv)[0] == '-' && eflag == FALSE)
    for(temp = argv[0]+1; *temp != '\0'; temp++)
      switch(toupper(*temp))
      {
        case 'V':
          vflag = TRUE;
          break;
        case 'C':
          cflag = TRUE;
          break;
        case 'L':
          lflag = TRUE;
          break;
        case 'N':
          nflag = TRUE;
          break;
        case 'H':
          hflag = TRUE;
          break;
        case 'Y':
          yflag = TRUE;
          break;
        case 'E':
          eflag = TRUE;
          break;
        case 'F':
          fflag = TRUE;
          break;
        case 'X':
          xflag = TRUE;
          break;
        case 'P':
          pflag = TRUE;
          break;
        case 'S':
          sflag = TRUE;
          break;
        default:
          error(OPT_ERR,NULL);
```

(Continued on next page)

Fgrep Listing

(Listing Continued, text begins on page 46)

```
    }

/* "pflag" can only be TRUE if the following flags are FALSE */

if(vflag == TRUE || cflag == TRUE || lflag == TRUE ||
    xflag == TRUE || sflag == TRUE)
  pflag = FALSE;

/* Check for string (or string file) argument */

  if(!argc)
    error(CMD_ERR,NULL);

  /* Build the "go" transitions. */

  bd_go(*argv++);
  argc--;

  /* Build the "failure" and "move" transitions */

  bd_move();

  /* Process each of the input files if not "stdin". */

  if(argc < 2)
    hflag = TRUE;
  if(!argc)
  {
    if(proc_file(NULL,FALSE) == TRUE && match_flag == FALSE)
      match_flag = TRUE;
  }
  else
    while(argc--)
      if(proc_file(*argv++,TRUE) == TRUE && match_flag == FALSE)
        match_flag = TRUE;

  /* Return status to the parent process. Status is zero if any
   * matches are found, 1 if none. */

  if(match_flag == TRUE)
    exit(0);
  else
    exit(1);
}

/*** Functions and Procedures ***/

/* PROC_FILE() - Run the FSA on the input file "in_file". Returns
 *               TRUE if a match was found, FALSE otherwise.
 */

BOOL proc_file(in_file,prt_flag)
char *in_file;
BOOL prt_flag;
{
  char buffer[MAX_LINE],        /* Input string buffer */
       *nl,
       *index(),
       *stoupper(),
       *fgets();

  long line_cnt = 0L,     /* Line counter */
       mtch_cnt = 0L;     /* Matched line counter */
  BOOL mtch_flag,         /* Matched line flag */
       run_fsa();
  FILE *in_fd,
       *fopen();
  void error();

  if(in_file != NULL)   /* A file was specified as the input */
  {
    if(!(in_fd = fopen(in_file,"r")))
      error(INP_ERR,in_file);
  }
  else
    in_fd = stdin;
```

(Continued on page 60)

Lifeboat.™

C is the language. Lifeboat™ is the source.

Productivity Tools from the Leading Publisher of C Programs.™

The Lattice® C Compiler

The cornerstone of a program is its compiler; it can make the difference between a good program and a great one. The Lattice C compiler features:

- Full compatibility with Kernighan and Ritchie's standards
- Four memory model options for control and versatility
- Automatic sensing and use of the 8087 math chip
- Choose from the widest selection of add-on options
- Renowned for speed and code quality
- Superior quality documentation

"Lattice C produces remarkable code...the documentation sets such a high standard that others don't even come close...in the top category for its quick compilation and execution time and consistent reliability."

Byte Magazine

Lattice Library source code also available.

Language Utilities

Pfix 86/Pfix 86 Plus — dynamic and symbolic debuggers respectively, these provide multiple-window debugging with breakpointing capability.
Plink 86 — a two-pass overlay linkage editor that helps solve memory problems.
Text Management Utilities — includes GREP (searches files for patterns), DIFF (differential text file comparator), and more.
LMK (UNIX "make") — automates the construction of large multi-module products.
Curses — lets you write programs with full screen output transportable among all UNIX, XENIX and PC-DOS systems without changing your source code.
BASTOC — translates MBASIC or CBASIC source code directly to Lattice C source code.
C Cross Reference Generator — examines your C source modules and produces a listing of each symbol and where it is referenced.

Editors

Pmate — a customizable full screen text editor featuring its own powerful macro command language.
ES/P for C — C program entry with automatic syntax checking and formatting.
VEDIT — an easy-to-use word processor for use with V-PRINT.
V-PRINT — a print formatting companion for VEDIT.
CVUE — a full-screen editor that offers an easy way to use command structure.
EMACS — a full screen multi window text editor.
Fast/C — speeds up the cycle of edit-compile-debug-edit-recompile.

Graphics and Screen Design

HALO — one of the industry's standard graphics development packages. Over 150 graphics commands including line, arc, box, circle and ellipse primitives. The **10 Fontpack** is also available.
Panel — a screen formatter and data entry aid.
Lattice Window — a library of subroutines allowing design of windows.

Functions

C-Food Smorgasbord — a tasty selection of utility functions for Lattice C programmers; includes a binary coded decimal arithmetic package, level 0 I/O functions, a Terminal Independence Package, and more.
Float-87 — supports the 8087 math chip to boost the speed of floating-point calculations.
The Greenleaf Functions — a comprehensive library of over 200 routines.
The Greenleaf Comm Library — an easy-to-use asynchronous communications library.
C Power Packs — sets of functions useful for a wide variety of applications.
BASIC C — This library is a simple bridge from IBM BASIC to C.

Database Record Managers

Phact — a database record manager library of C language functions, used in the creation and manipulation of large and small databases.
Btrieve — a sophisticated file management system designed for developing applications under PC-DOS. Data can be instantly retrieved by key value.
FABS — a Fast Access Btree Structure function library designed for rapid, keyed access to data files using multipath structures.
Autosort — a fast sort/merge utility.
Lattice dB-C ISAM — a library of C functions that enables you to create and access dBase format database files.

Cross-Compilers

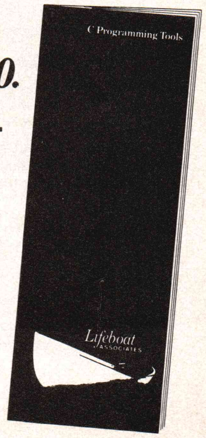
For programmers active in both micro and mini environments we provide advanced cross-compilers which product Intel 8086 object modules. All were developed to be as functional — and reliable — as the native compilers. They are available for the following systems:

VAX/VMS, VAX/UNIX, 68K/UNIX-S, 68K/UNIX-L

Also, we have available:
Z80 Cross-Compiler for MS- and PC-DOS — produces Z80 object modules in the Microsoft relocatable format.

New Products

Run/C — finally, a C interpreter for all levels of C Programmers.
C Sprite — a symbolic debugger with breakpoint capability.



Call LIFEBOAT: 1-800-847-7078. In NY, 1-212-860-0300.

YES! Please rush me the latest FREE Lifeboat™ catalog of C products.

*Name*__________ *Title*__________

*Company Name*__________ *Business Phone*__________

*Address*__________

Please check one of the following categories:

☐ *Dealer/Distributor* ☐ *End User* ☐ *Other*__________

Return Coupon to: Lifeboat™ Associates
1651 Third Avenue, New York, NY 10128

DD

Circle no. **125** on reader service card.

Fgrep Listing

(Listing Continued, text begins on page 46)

```
   /* Read in a line at a time for processing */

   while(fgets(buffer,MAX_LINE,in_fd))
   {
     if(nl = index(buffer,'\n'))   /* Remove newline */
       *nl = '\0';
#ifdef CP/M
     if(fflag == FALSE || yflag == TRUE)
       stoupper(buffer);
#else
     if(yflag == TRUE)
       stoupper(buffer);
#endif
     line_cnt++;            /* Increment the line counter */
     if((mtch_flag = run_fsa(buffer)) == TRUE)
       mtch_cnt++;          /* Increment matched line counter */
     if(cflag == FALSE && lflag == FALSE && sflag == FALSE &&
         ((mtch_flag == TRUE && vflag == FALSE) ||
         (mtch_flag == FALSE && vflag == TRUE)))
     {
       if(hflag == FALSE && prt_flag == TRUE)
         printf("%s: ",in_file);
       if(nflag == TRUE)
         printf("%05ld: ",line_cnt);
       puts(buffer);
     }
   }
   if(lflag == TRUE && mtch_cnt > 0)
     printf("%s\n",in_file);
   else if(cflag == TRUE && sflag == FALSE)
     printf("%ld\n",mtch_cnt);
   if(in_file != NULL)
     fclose(in_fd);
   if(mtch_cnt)             /* Match found */

     return TRUE;
   else                     /* No match found */
     return FALSE;
}

/* RUN_FSA() - Run the finite state automaton with string "str"
 *               as input. Return TRUE if match, FALSE otherwise.
 */

BOOL run_fsa(str)
register char *str;
{
  register FSA *st_ptr;
  char *message = NULL;
  BOOL msg_flag = FALSE;
  FSA *go(),
      *move();

  st_ptr = NULL;           /* Initialize FSA */
  if(xflag == FALSE)
  {
    /* Process the next input character in the string */

    while(*str)
    {
      st_ptr = move(st_ptr,*str);

      /* Print terminal state message and update FSA */

      if(st_ptr == 0 && message)
      {
        printf("--> %s\n",message);
        message = NULL;
        st_ptr = move(st_ptr,*str);
      }
      str++;
      if(st_ptr)
        if(st_ptr->out_str)      /* Terminal state? */
          if(pflag == TRUE)
          {
            /* Save terminal state message */

            message = st_ptr->out_str;
            msg_flag = TRUE;
```

```
        }
        else
          return TRUE;
    }
    if(message)  /* Print any remaining terminal state message */
      printf("--> %s\n",message);

      return msg_flag;
    }
    else  /* Match exact lines only */
    {
      while(*str)
      {
        st_ptr = go(st_ptr,*str++);
        if(!st_ptr || st_ptr == &FAIL_STATE)
          return FALSE;    /* Line not matched */
      }
      return TRUE;         /* Exact line matched */
    }
}

/* GO() - Process "litchar" and return a pointer to the FSA's
 *        corresponding "go" transition state. If the character
 *        is not in the FSA state's "go" transition list, then
 *        return a pointer to FAIL_STATE.
 */

FSA *go(st_ptr,litchar)
FSA *st_ptr;
register char litchar;
{
  register TRANSITION *current;

  /* If State 0, then access separate State 0 data structure of
   * the FSA. Note that there are no failure states defined for
   * any input to FSA State 0.
   */

  if(!st_ptr)
    return MZ[litchar];
  else
  {
    /* Point to the head of the linked list of "go" transitions
     * associated with the state.
     */

    current = st_ptr->go_ls;

    /* Transverse the list looking for a match to the input
     * character.
     */

    while(current)
    {
      if(current->lchar == litchar)
        break;
      current = current->next_el;

    }

    /* Null value for "current" indicates end of list was reached
     * without having found match to input character.
     */

    return current ? current->nextst_ptr : &FAIL_STATE;
  }
}

/* MOVE() - Process "litchar" and return a pointer to the FSA's
 *          corresponding "move" transition state.
 */

FSA *move(st_ptr,litchar)
FSA *st_ptr;
register char litchar;
{
  register TRANSITION *current;

  /* If State 0, then access separate State 0 data structure of
   * the FSA.
   */
```

(Continued on next page)

Fgrep Listing

(Listing Continued, text begins on page 46)

```
if(!st_ptr)
  return MZ[litchar];
else
{
  /* Point to the head of the linked list of "move" transitions
   * associated with the state.
   */

  current = st_ptr->mv_ls;

  /* Transverse the list looking for a match to the input
   * character.
   */

  while(current)
  {
    if(current->lchar == litchar)
      break;
    current = current->next_el;
  }

  /* Null value for "current" indicates end of list was reached
   * without having found match to input character. The
   * returned pointer is then to State 0.
   */

     return current ? current->nextst_ptr : NULL;
  }
}

/* BD_GO() - Build the "go" transitions for each state from the
 *           command-line arguments.
 */

void bd_go(str)
char *str;
{
   register char litchar;
   char *nl,
        buffer[MAX_LINE],        /* Input string buffer */
        *stoupper(),
        *fgets(),
        *index();
   FILE *str_fd,
        *fopen();
   void error(),
        enter();

   /* Initialize FSA State 0 "go" transition array so that every
    * invocation of "go()" with "state" = 0 initially returns a
    * pointer to FAIL_STATE.
    */

   for(litchar = 1; litchar <= 127; litchar++)
     MZ[litchar] = &FAIL_STATE;

   /* If the -f option was selected, get the newline-separated
    * strings from the file "str" one at a time and enter them
    * into the FSA. Otherwise, enter the string "str" into the
    * FSA.
    */

   if(fflag == TRUE)
   {
     if(!(str_fd = fopen(str,"r")))
       error(STR_ERR,str);

     while(fgets(buffer,MAX_LINE,str_fd))
     {
       if(nl = index(buffer,'\n'))         /* Remove the newline */
         *nl = '\0';
       if(yflag == TRUE)
         stoupper(buffer);
       enter(buffer);
     }
     fclose(str_fd);
```

```
    }
    else
    {
      if(yflag == TRUE)
        stoupper(buffer);
      enter(str);
    }

    /* For every input character that does not lead to a defined
     * "go" transition from FSA State 0, set the corresponding
     * element in the State 0 "go" transition array to indicate
     * a "go" transition to State 0.
     */

    for(litchar = 1; litchar <= 127; litchar++)
      if(MZ[litchar] == &FAIL_STATE)
        MZ[litchar] = NULL;
}

/* ENTER() - Enter a string into the FSA by running each
 *           character in turn through the current partially-
 *           built FSA. When a failure occurs, add the remainder
 *           of the string to the FSA as one new state per
 *           character. (Note that '\0' can never be a valid
 *           character - C uses it to terminate a string.)
 */

void enter(str)
char *str;
{
  FSA *s,
      *go(),
      *create();
  TRANSITION *current,
             *insert();
  char *strsave();
  register char *temp;
  register FSA *st_ptr = NULL;  /* Start in FSA State 0 */
  register FSA *nextst_ptr;
  void error();

  /* Run each character in turn through partially-built FSA until
   * a failure occurs.
   */

  temp = str;
  while((s = go(st_ptr,*temp)) != &FAIL_STATE)
  {
    temp++;
    st_ptr = s;

  }

  /* Process the remainder of the string */

  while(*temp)
  {
    /* If a new state, then create a new state and insert
     * transition character and "go" transition in current
     * state. (Note special case of FSA State 0.)
     */

    if(!st_ptr)
      nextst_ptr = MZ[*temp++] = create();
    else if(!(current = st_ptr->go_ls))
    {
      nextst_ptr = create();
      st_ptr->go_ls = insert(nextst_ptr,*temp++);
    }
    else
    {
      /* ... or it was the character that the FSA returned a
       * "failure" for. Find the tail of the current state's list
       * of "go" transitions, create a new state and append it
       * to the current state's "go" list.
       */

      while(current->next_el)
        current = current->next_el;
      nextst_ptr = create();
      current->next_el = insert(nextst_ptr,*temp++);
    }
    st_ptr = nextst_ptr;
  }
```

(Continued on next page)

Fgrep Listing

(Listing Continued, text begins on page 46)

```
     /* Make string terminal state's output message */

     st_ptr->out_str = strsave(str);
}

/* INSERT() - Create a new "go" transition and return a pointer
 *            to it.
 */

TRANSITION *insert(st_ptr,litchar)
FSA *st_ptr;
char litchar;
{
  TRANSITION *current;
  char *malloc();
  void error();

  if(!(current = (TRANSITION *)malloc(sizeof(TRANSITION))))
    error(MEM_ERR,NULL);
  current->lchar = litchar;
  current->nextst_ptr = st_ptr;
  current->next_el = NULL;
  return current;
}

/* CREATE() - Create an FSA state and return a pointer to it. */

FSA *create()
{
  FSA *st_ptr;
  char *malloc();
  void error();

  if(!(st_ptr = (FSA *)malloc(sizeof(FSA))))
    error(MEM_ERR,NULL);
  st_ptr->go_ls = st_ptr->mv_ls = NULL;
  st_ptr->fail_state = NULL;
  st_ptr->out_str = NULL;
  return st_ptr;
}

/* BD_MOVE() - Build the "failure" and "move" transitions for
 *             each state from the "go" transitions.
 */

void bd_move()
{
  register char litchar;
  register FSA *r,         /* Temporary FSA state pointers */
               *s,
               *t;
  FSA *go(),
      *move();
  TRANSITION *current,
             *insert();
  QUEUE *first,            /* Pointer to head of queue */
        *last;             /* Pointer to tail of queue */
  void add_queue(),
       delete_queue();

  last = first = NULL;     /* Initialize the queue of FSA states */

  /* For each input character with a "go" transition out of FSA
   * State 0, add a pointer to the "go" state to the queue. Note
   * that this will also serve as the "move" transition list for
   * State 0.

   */

  for(litchar = 1; litchar <= 127; litchar++)
    if(s = go(NULL,litchar))
      add_queue(&first,&last,s);

  /* While there are still state pointers in the queue, do ... */

  while(first)
```

(Continued on page 66)

PROGRAMMER DEVELOPMENT TOOLS

TURBO PASCAL AND UTILITIES

	List	Ours
Turbo PASCAL by Borland International .... *Sale*	70	49
Turbo PASCAL *w/8087 or BCD* .... *Sale*	110	89
Turbo PASCAL *w/8087 & BCD* .... *Sale*	125	99
Btrieve by Softcraft	250	199
FirsTime for Turbo by Spruce Technology	75	69
Multi-Halo Graphics by Media Cybernetics	250	199
Screen Sculptor by Software Bottling	125	109
Turbo ASYNCH by Blaise Computing	100	89
Turbo GRAPHICS TOOLBOX by Borland Int'l.	55	49
Turbo POWER TOOLS by Blaise Computing . *New*	100	89
Turbo TOOLBOX by Borland Int'l	55	49
Turbo TUTOR by Borland Int'l	35	29
TurboPower Util *w/source* by TurboPower Sftwr	95	89
TurboWindow by MetaGraphics	55	49
XTC *Text Editor* by Wendin	99	89

BASIC LANGUAGE

	List	Ours
BetterBASIC by Summit Software	200	169
8087 Math Support	99	89
Run-time Module	250	239
Professional BASIC by Morgan Computing	99	89
8087 Math Support	50	47
True Basic from Addison-Wesley ...... *New*	150	129
Run-time Module	500	459

OTHER LANGUAGES

	List	Ours
8088 Assembler *w/Z-80 Translator* by 2500 AD	100	89
APL*PLUS/PC by STSC	595	449
Golden Common LISP by Gold Hill	495	Call
Janus/ADA by R&R Software	900	699
MASM-86 *w/utilities* by Microsoft	150	109
Microsoft Fortran .... *Links with MS C ver 3*	350	239
Microsoft Pascal .... *Links with MS C ver 3*	300	219
Modula-2/86 by Logitech	495	439
Pocket APL by STSC .... *with rebate offer*	95	89
Prolog VMI by Automata Design Associates . *New*	100	89
Prolog VML by Automata Design Associates *New*	300	269
Prolog VMA by Automata Design Associates *New*	500	439
Prolog-86 by Solution Systems	125	Call
RM/Fortran By Ryan-McFarland	595	439

OTHER LANGUAGE SUPPORT PRODUCTS

	List	Ours
APL2C by Lauer Software ..... *Interfaces APL to C*	150	139
Btrieve By SoftCraft	250	199
Blaise Tools *for Pascal*	Call	Call
FORTRAN Libraries by Alpha Computer Service	Call	Call
Sci Subroutine Lib *for Fortran or Basic*	175	139

OTHER PRODUCTS

	List	Ours
Advanced Trace-86 by Morgan Computing	175	149
Codesmith-86 *Debugger* by Visual Age	145	129
Polytron Products .... *We Carry a Full Line*	Call	Call
Profiler by DWB Associates	125	89
Rtrieve by Softcraft	85	79
Xtrieve by Softcraft	195	169

Periscope Symbolic Debugger
by Data Base Decisions

Write-protect memory board and breakout switch allows instant recovery from runaway code. Provides on-line help, windowing, extensive breakpoints, dual monitor support and more.

List Price $295 Our Price **$269**

PHOENIX PRODUCTS

SUMMERTIME SALE!
In stock and ready for immediate shipping.

	List	Ours
Pasm86 *High Performance Macro Assembler*	295	195
Pfinish *Performance Analyzer*	395	269
Pfix-86 Plus *Symbolic Debugger for Plink-86*	395	269
Plink-86 *Overlay Linker*	395	269
Pmaker *Program Development Manager*	195	119
Pmate *Macro Text Editor*	225	145
Pre-C *Lint Utility*	395	269

Dealer Inquiries Invited

C LANGUAGE

	List	Ours
C-terp *C Interpreter* by Gimpel Software	300	269
Computer Innovations C-86 Compiler	395	299
DeSmet C Compiler with Debugger	159	145
Instant C by Rational Systems .... *Sale*	500	399
Lattice C Compiler from Lattice	500	349
Lattice C from Lifeboat .... *Ltd Qty Special*	500	275
Mark Williams MWC-86 w/Source Debugger . *Sale*	495	379
Microsoft C Compiler *version 3* .... *Sale*	395	259
Wizard C Compiler by Wizard Systems ...... *Sale*	450	359
Xenix Development System by SCO	1350	1099

MACINTOSH C AND UTILITIES

	List	Ours
c-tree by FairCom .... *New version*	395	359
DeSmet/Ouye C Compiler .... *New*	150	139
Mac C by Consulair .... *Call for Details*	Call	Call
Megamax C *compiler for MacIntosh*	295	239

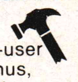
MICROSOFT C AND UTILITIES

	List	Ours
Microsoft C Compiler *version 3* .... *Sale*	395	259
Blaise C Tools	125	109
Blaise C Tools 2	100	89
C Power Packs from Software Horizons	Call	Call
C-terp by Gimpel Software	300	269
C Utility Library by Essential Software	185	139
Greenleaf C Functions Library ...... *New version*	185	139
Greenleaf Comm Library	185	139
The HAMMER by OES Systems .... *New*	195	179
Multi-Halo Graphics by Media Cybernetics	250	199
PANEL *Screen Designer* by Roundhill	295	234
Windows for C by Vermont Creative Software	195	139

C UTILITIES

	List	Ours
Basic_C Library by C Source	175	139
Blaise C Tools	125	109
Blaise C Tools 2	100	89
Btrieve by SoftCraft	250	199
C Power Paks From Software Horizons	Call	Call
c-tree by FairCom .... *New version*	395	359
C Utility Library by Essential Software	185	139
ESP for C by Bellesoft .... *Sale*	349	229
GraphiC by Scientific Endeavors	250	209
Greenleaf C Functions Library ...... *New Version*	185	139
Greenleaf Comm Library	185	139
MetaWINDOWS by Metagraphics	150	139
Multi-Halo Graphics by Media Cybernetics	250	199
PANEL *Screen Designer* by Roundhill	295	234
PC Lint by Gimpel Software	100	89
Scientific Subroutine Lib *for C* by Peerless	175	139
Windows For C by Vermont Creative Software	195	139

The **HAMMER** C Library by OES



This excellent new C library is designed for creating end-user interfaces. There are functions for creating 123-like menus, managing the screen, creating input forms, prompting for inputs and more. No royalties and includes source code.

List Price $195 Our Price **$179**

LATTICE PRODUCTS

All products receive support and updates directly from Lattice. Lattice C is in stock and ready for shipment.

	List	Ours
Lattice C Compiler from Lattice	500	349
C-Food Smorgasbord	150	119
C-Sprite *Program Debugger*	175	149
Curses *Screen Manager*	125	109
dBC *C Interface for dBase II or III Files*	250	209
dBC *with source code*	500	439
LMK *Make Facility*	195	159
Text Mgmt Utils *(GREP/DIFF/ED/WC/Extract/Build)*	120	105

TEXT EDITORS

	List	Ours
Brief By Solution Systems	195	Call
Epsilon *Emacs-like Text Editor* by Lugaru	195	179
ESP for C *or* **Pascal** by Bellesoft	Call	Call
ESP for C and Pascal by Bellesoft .... *Sale*	399	279
FirsTime for C by Spruce Technology	295	239
FirsTime for MS Pascal by Spruce Technology	245	199
Vedit by Compuview	Call	Call
XTC *Text Editor* by Wendin	99	89

PC

Programmer's Connection

136 Sunnyside Street Hartville, Ohio 44632 (216) 877-3781 (In Ohio)

U.S.: **1-800-336-1166** Canada: **1-800-225-1166**

Call For Our Catalog

MasterCard VISA

Account is charged when order is shipped
Prices are subject to change without notice
We accept qualified Corporate Accounts and P.O.'s

Circle **no. 120** on reader service card.

Fgrep Listing (Listing Continued, text begins on page 46)

```
  {
    /* Remove State "r" pointer from the head of the queue. */

    r = first->st_ptr;
    delete_queue(&first);

    /* Skip (terminal) state with no "go" transitions" */

    if(!r->go_ls)
      continue;

    /* Make "move" transition list for terminal state same as its
     * "go" transition list.
     */

    if(r->out_str)
      r->mv_ls = r->go_ls;

    /* For every input to State "r" that has a "go" transition to
     * State "s", do ...
     */

    for(litchar = 1; litchar <= 127; litchar++)
    {
      if((s = go(r,litchar)) != &FAIL_STATE)
      {
        /* If a defined "go" transition exists for State "r" on
         * on input "litchar", add a pointer to State "s" to the
         * end of the queue.
         */

        add_queue(&first,&last,s);

        /* Calculate the "failure" transition of State "s" using
         * the following algorithm.
         */

        t = r->fail_state;
        while(go(t,litchar) == &FAIL_STATE)
          t = t->fail_state;
        s->fail_state = go(t,litchar);
      }
      else
      {
        /* ... otherwise set the pointer to State "s" to a
         * pointer to the precalculated "move" transition of
         * State "r"'s failure state on input "litchar".
         */

        s = move(r->fail_state,litchar);
      }

      /* Add State "s" as the "move" transition for State "r" on
       * input "litchar" only if it is not State 0 and "r" is not
       * a terminal state.
       */

      if(s && !r->out_str)
        if(!r->mv_ls)   /* First instance of the list? */
          current = r->mv_ls = insert(s,litchar);
        else            /* No, just another one ... */
          current = current->next_el = insert(s,litchar);
    }
  }
}

/* ADD_QUEUE() - Add an instance to the tail of a queue */

void add_queue(head_ptr,tail_ptr,st_ptr)
QUEUE **head_ptr,
      **tail_ptr;
FSA *st_ptr;
{
  QUEUE *pq;
  char *malloc();
  void error();

  /* Allocate the necessary memory and set the variables. */

  if(!(pq = (QUEUE *)malloc(sizeof(QUEUE))))
    error(MEM_ERR,NULL);
```

```
    pq->st_ptr = st_ptr;
    pq->next_el = NULL;

    if(!*head_ptr)          /* First instance of the queue? */
      *tail_ptr = *head_ptr = pq;
    else                    /* No, just another one ... */
      *tail_ptr = (*tail_ptr)->next_el = pq;
  }

  /* DELETE_QUEUE() - Delete an instance from the head of queue */

  void delete_queue(head_ptr)
  QUEUE **head_ptr;
  {
    *head_ptr = (*head_ptr)->next_el;
  }

  /* STRSAVE() - Save a string somewhere in memory */

  char *strsave(str)
  char *str;
  {
    int strlen();
    char *p,
         *malloc();
    void error();

    if(p = malloc(strlen(str) + 1))
      strcpy(p,str);
    else
      error(MEM_ERR,NULL);
    return p;
  }

  /* STOUPPER() - Map entire string pointed to by "str" to upper
   *              case.
   */

  char *stoupper(str)
  register char *str;
  {
    register char *temp;

    temp = str;
    while(*temp)
      *temp++ = toupper(*temp);
    return str;
  }

  /* ERROR() - Error reporting. Returns an exit status of 2 to the
   *           parent process.
   */

  void error(n,str)
  int n;
  char *str;
  {
    fprintf(stderr,"\007\n*** ERROR - ");
    switch(n)
   {
     case CMD_ERR:
       fprintf(stderr,"Illegal command line");
       break;
     case OPT_ERR:
       fprintf(stderr,"Illegal command line option");
       break;
     case INP_ERR:
       fprintf(stderr,"Can't open input file %s",str);
       break;
     case STR_ERR:
       fprintf(stderr,"Can't open string file %s",str);
       break;
     case MEM_ERR:
       fprintf(stderr,"Out of memory");
       break;
     default:
       break;
   }
   fprintf(stderr," ***\n\nUsage: fgrep [-vclnhyefxps]");
   fprintf(stderr," [strings] <files>\n");
   exit(2);
}

/*** End of FGREP.C ***/
```

End Listing

Bose-Nelson Sort

by Joe Celko

Sorting algorithms can be divided into two types: those that preserve the original sequence of the records (also called stable sorts) and those that don't. For example, if a stable alphabetic sort is applied twice to an address file, first on city names then on states, the result will be a list ordered by cities within states. The same file sorted with a non-stable sort would lose the city ordering on the second sort. Although the ability to use stable sorts in a series makes them look more attractive than the non-stable sorts, the non-stable sorts are faster.

All sorting algorithms consist of an exchange operation that swaps two items in the file or array and puts them into sorted order. The number of exchanges done by a sort determines how fast the sort runs—and this number varies with the existing order in the input file.

No sort algorithm is optimal under all conditions. This one is useful when stability is not a consideration or when parallel processing is possible.

This gives us a worst-case number of exchanges, a best-case number of exchanges, and an expected number of exchanges. For example, the Quicksort algorithm[1] has a best-case time for a file that is already sorted (most algorithms do quite well in this case) and it has a worst case for a file in reverse order. Its expected performance is (**n log2(n)**) exchanges, where **(n)** is the number of records. This makes Quicksort one of the best sorting algorithms known to us. But do we know just how well a sort can perform?

In theory, the fewest number of exchanges required to sort a file of **(n)** items is the ceiling of **log2(n!)** exchanges. For example, for a file of five items, we have **log2(5!) = log2(120) = 6.9068**, which rounds up to seven exchanges. The next question is whether such algorithms perform equally well for different numbers of records.

The answer is mixed: optimal algorithms exist for some values and not for others. No single algorithm seems to be minimal for all values. Many of the minimal algorithms are highly specialized. For example, Dr. H. B. Demuth discovered an algorithm for sorting five items—and only five—in seven exchanges.[2]

Can an algorithm be generalized for any fixed number of records and still come pretty close to being optimal? This question leads us to the P-operator discovered by R. C. Bose and R. J. Nelson.[3] This is a recursively defined function that generates exchange pairs for a fixed number of items.

The bad news is that the algorithm takes a long time to generate these exchange pairs; you really should not use it directly to sort a file. The good news is that you need generate the pairs only once, and you have a very good sort for a given number of items. An application of this might be sorting a poker or bridge hand in a

Joe Celko, 8833 Sunset Blvd #304, Los Angeles, CA 90069.

card-playing program.

The P-operator can be defined as ordered tuples of integers with a set of transformation rules that are applied left to right[4] The rules appear in the Figure (at right) and can be used to write a recursive program. The one given in Listing One (page 70) is credited to Dennis Allison. Another approach is to use an explicit stack and push quintuples, as shown in Listing Two (page 76). This second program will run faster due to the recursive procedure calls and lack of **printf** functions in C. Readers with accounts on CompuServe can obtain the source code and a short piece of documentation for Listing Two by accessing [70665, 1307].

What is happening in this confusing set of rules is that the original array is divided into two ordered sequences, which are merged into one ordered sequence. The two subsequences, in turn, are ordered by recursive calls to the same procedure. For five terms, the Bose-Nelson P-operator program will generate the following C program:

```
{
swap(1, 2);
swap(4, 5);
swap(3, 5);
swap(3, 4);
swap(1, 4);
swap(1, 3);
swap(2, 5);
swap(2, 4);
swap(2, 3);
}
```

You can check this program by taking five playing cards and dealing them face down in a row. Turn over cards 1 and 2; if they are out of order, swap them and turn them face down again. Repeat this procedure until you have made all of the swaps in the program then turn over all of the cards. They should be in sorted order now.

It is worth noting that this program has nine steps and that we already know seven is the optimal number. So, contrary to earlier beliefs,[5] the Bose-Nelson does not produce optimal sorts. (It is left as an exercise for the reader to see how well the Bose-Nelson sort performs compared

```
P(1, x, y) => Swap(x, y);

P(2, i, 1) => Null
P(2, i, x) => P(2,i,a) P(2,(i+a),(x-a)) P(3,i,a,(i+a), (x-a))

P(3, i, 0, j, y) => Null
P(3, i, x, j, 0) => Null
     (These tuples should never appear, but they are given for completeness.)
P(3, i, 1, j, 1) => P(1, i, j)
P(3, i, 1, j, 2) => P(1, i, (j+1)) P(1, i, j)
P(3, i, 2, j, 1) => P(1, i, j) P(1, (i+1), j)
P(3, i, x, j, y) => P(3, i, a, j, b) P(3, (i+a), (x-a), (j+b), (y-b))
    P(3, (i+a), (x-a), j, b)

where  a =  [x/2]
       b =  [(y+1)/2] if (x is even)
            [y/2] if (x is odd)
```

Figure
Transformation Rules for Ordered Tuples

with the optimal values.) However, the swap pairs do not depend on the previous results, which makes it easy to use them for hard-wired sorting machines. It also makes it possible to use them in parallel processor computers: in the above five-item sort, the calls to swap(1, 2) and swap(4, 5) can be done at the same time.

References

[1] Hoare, C. A. R., "Algorithm 64: Quicksort," *Communications of the ACM*, 4 (1961) p. 321.

[2] Demuth, H. B., PhD thesis, Stanford University, 1956, pp. 41–43.

[3] Bose and Nelson, "A Sorting Problem," *JACM*, Vol. 9 (1962) pp. 282–296.

[4] [RIC 1972] Rich, Robert P., Internal Sorting Methods Illustrated with PL/I Programs, Prentice Hall, 1972.

[5] Knuth, Donald, *Sorting and Searching*, Addison-Wesley, 1973.

DDJ

Bose-Nelson Sort (Text begins on page 68)

Listing One

```
* Recursive version of Bose-Nelson sort */
#include "STDIO.H"
#include "atou.c" /* ASCII to unsigned integer */

int count = 0;

main (argc, argv)
int argc;
char argv[];
{ int n;
  if (argc < 2)
     { printf ("USAGE: bose n >outfile\n");
       exit();
     }
  else
     { n = (atou(*(ap = *++argv) == '-'? ap+1: ap);
       bosesort (1, n);
       printf ("There were %d swaps\n", count);
     }
}

bosesort(i, j)
int i, j;
{  int m;
   if (n > 1)
      { m = 1 + (j-1+1)/2 - 1;
        bosesort (i, m);
        bosesort ((m+1), j);
        bosemerge (i, m, (m+1), j);
      }
}

bosemerge (i1, i2, j1, j2)
int i1, i2, j1, j2;
{ if ((i2 == i1) && (j2 == j1))
     { printf ("swap(%d, %d);\n", i1, j1);
       count += 1;
     }
  else if ((i2 == (i1 + 1)) && (j2 == j1))
     { printf ("swap(%d, %d);\n", i1, j1);
       printf ("swap(%d, %d);\n", i2, j1);
       count += 2;
```

(Continued on page 72)

Listing One

```
   }
 else if ((i2 == il) && (j2 == (jl+l))
   { printf ("swap(%d, %d);\n", il, j2);
     printf ("swap(%d, %d);\n", il, jl);
     count += 2;
   }
 else
   { i_mid = il + (i2-il+l)/2 -l;
     if (even(i2 - il+l) && (i2-il != j2-jl))
        j_mid = jl + (j2-jl+l)/2;
     else
        j_mid = jl + (j2-jl+l)/2 - l;

     bosemerge (il, i_mid, jl, j_mid);
     bosemerge (i_mid+l, i2, j_mid+l, j2);
     bosemerge (i_mid+l, i2, jl, j_mid);

   }
}

even (x) int x; { return (l - (l & x));}
```

End Listing One

(Listing Two begins on page 74)

Listing Two

```
/* non-recursive version of Bose-Nelson sort */
#include "STDIO.H"
#include "atou.c" /* ASCII to unsigned integer */
#define MAXSTACK 2500  /* pick a high number */

int count = 0;
int top = 0;
int stack[MAXSTACK];

main (argc, argv)
int argc;
char argv[];
{ int n;
  if (argc < 2)
     { printf ("USAGE: bose n >outfile\n");
       exit();
     }
  else
     { n = (atou(*(ap = *++argv) == '-'? ap+1: ap);
       bosesort (n);
       printf ("There were %d swaps\n", count);
     }
}


 bosesort(n)
 int n;
 { int a, b, i, j, x, y;
   printf ("{\n");
   push5 (2,1,n,0,0);
   while (top >0)
    switch (stack[top])
    {
    case 0: { printf("}\n");
              top = 0;
            }
            break;

    case 1: { printf("swap(%d,%d);\n", stack[top-1], stack[top-2]);
              top -= 3;
              count++;
            }
            break;

    case 2: { i = stack[top-1];
              x = stack[top-2];
       /* these two variables are defined for easy reading */
              top -= 3;
              if ( x != 1)
                 { a = (x/2);
                   push3(3, i, a, (i+a), (x-a));
                   push3(2, (i+a), (x-a));
                   push3(2, i, a);
```

(Continued on page 76)

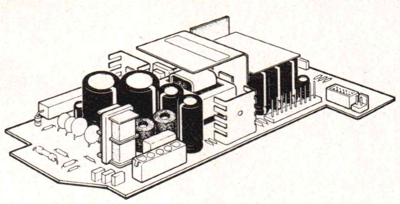
ALL SALES ARE MADE SUBJECT TO THE TERMS OF OUR 90 DAY LIMITED WA... A COPY OF THIS WARRANTY IS AVAILABLE FREE, ON REQUEST.

DIGITAL RESEARCH COMPUTERS

(214) 225-2309

Switching Power Supply!

+ 5VDC - 8 Amps
+12VDC - 5 Amps
-12VDC - 1 Amp
(81 WATTS MAX)

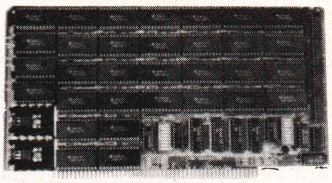


$29.95 ea. | 4 FOR $99 | ADD $2 EA. UPS

BRAND NEW UNITS, MFG. BY BOSCHERT FOR HEWLETT PACKARD! PERFECT FOR SMALL COMPUTER AND DISK DRIVE APPLICATIONS #XL81-5630. INPUT 110V/220V, 50/60 HZ. NOMINAL OUTPUTS: +5VDC @ 8A, +12VDC @ 5A, -12VDC @ 1A. TOTAL MAX OUTPUT. MUST BE LIMITED TO 81 WATTS TOTAL! (WITH PIN OUT SHEET.) ORIGINAL FACTORY BOXED.

64K S100 STATIC RAM

$119.00 KIT



LOW POWER!

150 NS ADD $10

BLANK PC BOARD WITH DOCUMENTATION $49.95

SUPPORT ICs + CAPS $17.50

FULL SOCKET SET $14.50

FULLY SUPPORTS THE NEW IEEE 696 S100 STANDARD (AS PROPOSED)

FOR 56K KIT $105

ASSEMBLED AND TESTED ADD $50

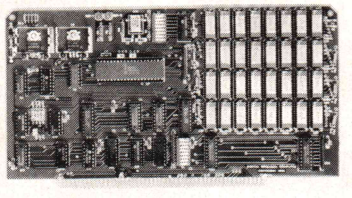
PRICE CUT!

FEATURES:
* Uses new 2K x 8 (TMM 2016 or HM 6116) RAMs.
* Fully supports IEEE 696 24 BIT Extended Addressing.
* 64K draws only approximately 500 MA.
* 200 NS RAMs are standard. (TOSHIBA makes TMM 2016s as fast as 100 NS. FOR YOUR HIGH SPEED APPLICATIONS.)
* SUPPORTS PHANTOM (BOTH LOWER 32K AND ENTIRE BOARD).
* 2716 EPROMs may be installed in any of top 48K.
* Any of the top 8K (E000 H AND ABOVE) may be disabled to provide windows to eliminate any possible conflicts with your system monitor, disk controller, etc.
* Perfect for small systems since BOTH RAM and EPROM may co-exist on the same board.
* BOARD may be partially populated as 56K.

256K S-100 SOLID STATE DISK SIMULATOR!

WE CALL THIS BOARD THE "LIGHT-SPEED-100" BECAUSE IT OFFERS AN ASTOUNDING INCREASE IN YOUR COMPUTER'S PERFORMANCE WHEN COMPARED TO A MECHANICAL FLOPPY DISK DRIVE.

PRICE CUT!



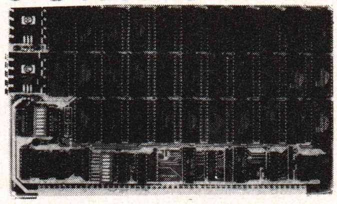
FEATURES:
* 256K on board, using + 5V 64K DRAMS.
* Uses new Intel 8203-1 LSI Memory Controller.
* Requires only 4 Dip Switch Selectable I/O Ports.
* Runs on 8080 or Z80 S100 machines.
* Up to 8 LS-100 boards can be run together for 2 Meg. of On Line Solid State Disk Storage.
* Provisions for Battery back-up.
* Software to mate the LS-100 to your CP/M* 2.2 DOS is supplied.
* The LS-100 provides an increase in speed of up to 7 to 10 times on Disk Intensive Software.
* Compare our price! You could pay up to 3 times as much for similar boards.

BLANK PCB (WITH CP/M* 2.2 PATCHES AND INSTALL PROGRAM ON DISKETTE) $69.95 (8203-1 INTEL $29.95)

(ADD $50 FOR A&T) **$149.00** (FULL 256K KIT) #LS-100

64K SS-50 STATIC RAM

$99.95 (48K KIT)



LOW POWER!

RAM OR EPROM!

BLANK PC BOARD WITH DOCUMENTATION $52

SUPPORT ICs + CAPS $18.00

FULL SOCKET SET $15.00

56K KIT $109

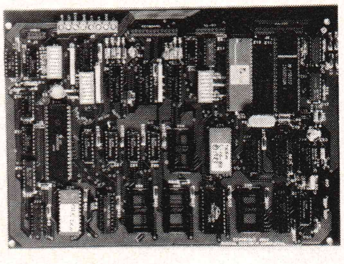
64K KIT $119

ASSEMBLED AND TESTED ADD $50

FEATURES:
* Uses new 2K x 8 (TMM 2016 or HM 6116) RAMs.
* Fully supports Extended Addressing.
* 64K draws only approximately 500 MA.
* 200 NS RAMs are standard. (TOSHIBA makes TMM 2016s as fast as 100 NS. FOR YOUR HIGH SPEED APPLICATIONS.)
* Board is configured as 3-16K blocks and 8-2K blocks (within any 64K block) for maximum flexibility.
* 2716 EPROMs may be installed anywhere on Board.
* Top 16K may be disabled in 2K blocks to avoid any I/O conflicts.
* One Board supports both RAM and EPROM.
* RAM supports 2MHZ operation at no extra charge!
* Board may be partially populated in 16K increments.

THE NEW ZRT-80 CRT TERMINAL BOARD!

A LOW COST Z-80 BASED SINGLE BOARD THAT ONLY NEEDS AN ASCII KEYBOARD, POWER SUPPLY, AND VIDEO MONITOR TO MAKE A COMPLETE CRT TERMINAL. USE AS A COMPUTER CONSOLE, OR WITH A MODEM FOR USE WITH ANY OF THE PHONE-LINE COMPUTER SERVICES.



FEATURES:
* Uses a Z80A and 6845 CRT Controller for powerful video capabilities.
* RS232 at 16 BAUD Rates from 75 to 19,200.
* 24 x 80 standard format (60 Hz).
* Optional formats from 24 x 80 (50 Hz) to 64 lines x 96 characters (60 Hz).
* Higher density formats require up to 3 additional 2K x 8 6116 RAMS.
* Uses N.S. INS 8250 BAUD Rate Gen. and USART combo IC.
* 3 Terminal Emulation Modes which are Dip Switch selectable. These include the LSI-ADM3A, the Heath H-19, and the Beehive.
* Composite or Split Video.
* Any polarity of video or sync.
* Inverse Video Capability.
* Small Size: 6.5 x 9 inches.
* Upper & lower case with descenders.
* 7 x 9 Character Matrix.
* Requires Par. ASCII keyboard.

FOR 8 IN. SOURCE DISK (CP/M COMPATIBLE) ADD $10

$89.95 #ZRT-80 (COMPLETE KIT, 2K VIDEO RAM)

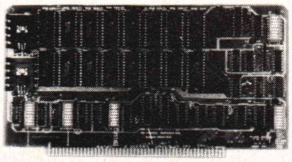
BLANK PCB WITH 2716 CHAR. ROM. 2732 MON. ROM $49.95

SOURCE DISKETTE - ADD $10

SET OF 2 CRYSTALS - ADD $7.50

32K S100 EPROM/STATIC RAM FOUR FUNCTION BOARD!

NEW! NEW!



EPROM II FULL EPROM KIT $69.95

A&T EPROM ADD $35.00

BLANK PC BOARD WITH DATA $39.95

SUPPORT IC'S PLUS CAPS $16

FULL SOCKET SET $15

We took our very popular 32K S100 EPROM Card and added additional logic to create a more versatile EPROM/RAM Board.

PRICES SLASHED!

FEATURES:
* This one board can be used in any one of four ways:
  A. As a 32K 2716 EPROM Board
  B. As a 32K 2732 EPROM Board (Using Every Other Socket)
  C. As a mixed 32K 2716 EPROM/2K x 8 RAM Board
  D. As a 32K Static RAM Board
* Uses New 2K x 8 (TMM2016 or HM6116) RAM's
* Fully Supports IEEE 696 Buss Standard (As Proposed)
* Supports 24 Bit Extended Adressing
* 200 NS (FAST!) RAM'S are standard on the RAM Kit
* Supports both Cromemco and North Star Bank Select
* Supports Phantom
* On Board wait State Generator
* Every 2K Block may be disabled
* Addressed as two separate 16K Blocks on any 64K Boundary
* Perfect for MP/M* Systems
* RAM Kit is very low power (300 MA typical)

32K STATIC RAM KIT — $99.95

For RAM Kit A&T — Add $40

Digital Research Computers

P.O. BOX 461565 • GARLAND, TEXAS 75046 • (214) 225-2309

TERMS: Add $3.00 postage. Orders under $15 add 75¢ handling. No C.O.D. We accept Visa and MasterCharge. Tex. Res. add 5-1/8% Tax. Foreign orders (except Canada) add 20% P & H. Orders over $50 add 85¢ for insurance.

*TM OF DIGITAL RESEARCH INC. (CALIF.)

WE ARE NOT ASSOCIATED WITH DIGITAL RESEARCH INC. (CALIF.) THE SUPPLIERS OF CPM SOFTWARE

```
                }
            }
            break;


    case 3: { i = stack[top-1];
              x = stack[top-2];
              j = stack[top-3];
              x = stack[top-4];
       /* these four variables are defined for easy reading */
              a = (x/2);
              if (even(x))
                 b = (y+1)/2;
              else
                 b = (y/2);
              top -= 5;
              if ((x == 1) && (y == 1))
                 push3 (1, i, j);
              else if ((x == 1) && (y == 2))
                      { push3(1, i, j);
                        push3(1, i, (j+1));
                      }
              else if ((x == 2) && (y == 1))
                      { push3(1, (i+1), j);
                        push3(1, i, j);
                      }
              else    { push5(3, (i+a), (x-a), j, b);
                        push5(3, (i+a), (x-a), (j+b), (y-b));
                        push5(3, i, a, j, b);
                      }
            }
            break;

     default:
            { printf ("FATAL: Error in stack\n");
              exit();
            }
            break;
   } /* end of while loop */
}/* end of main */

even (x)
int x;
{ return (1 - (1 & x));}

push3(a, b, c)
int a, b, c;
{ stack[++top] = c;
  stack[++top] = b;
  stack[++top] = a;
  if (top > MAXSTACK)
     { printf ("FATAL: stack overflow at %d\n", top);
```

```
        exit();
    }
}

push5(a, b, c, d, e)
int a, b, c;
{ stack[++top] = e;
  stack[++top] = d;
  stack[++top] = c;
  stack[++top] = b;
  stack[++top] = a;
  if (top > MAXSTACK)
     { printf ("FATAL: stack overflow at %d\n", top);
       exit();
     }
}
```

End Listings

Two TEX Implementations for the IBM PC

Richard Furuta and Pierre A. MacKay

"An experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TEX for formatting, and a language like Pascal for programming." —Donald Knuth, *The WEB System of Structured Documentation.*

TEX, as many readers will know, is Donald Knuth's system for typesetting technical text. The name is based on the Greek root in the word 'technical'; it is in no way associated with the state of Texas, and only remotely associated even with the word 'text'. The shibboleth of the true convert is the pronunciation of the final letter, which should sound rather like the 'ch' at the end of the name of J. S. Bach. The official derivation is from *tékhnē*='art' or 'craft'.

A system intended for the creation of beautiful books.

Knuth is a Professor of Computer Science at Stanford University and is best known for his ongoing series of books collectively entitled *The Art of Computer Programming*, perhaps the most important central reference source in Computer Science. TEX grew out of the need for a typesetting program that could be used to prepare new volumes in the series—in particular, the need for a program that would take care of the delicate formatting needed for mathematics typesetting. In the process of working out this specific problem, Knuth supplied just about every need that could be imagined for text typesetting. There are certain things TEX does not do; it has no built in graphics capabilities, and it does not provide for arbitrary rotations of characters and lines, but it should be noted that there are very few digital phototypesetters which could make use of such capacities even if they were offered. There are good general mechanisms within TEX which may someday be used to supply some of these special refinements if the output devices for them become readily available. For now, TEX offers precisely what Don Knuth claims for it in the preface to the *TEXbook*, "a system intended for the creation of beautiful books." TEX will be used, and is being used for everything from two-line business letters to railway schedules, but the heart of the program is book-production, and it can only be truly understood in that light.

Richard Furuta and Pierre A. MacKay, Univ. of Washington, Seattle WA.

An earlier version of this program, now distinguished as TEX78, was developed for the Stanford University DEC-10 system, using the SAIL programming language. Subsequently, TEX78 was translated into Pascal, and ported to a variety of mainframes. Based on this experience, the program was completely redesigned and reimplemented, and it is this newest version of the program that is today called TEX.

One of the commonest criticisms of TEX78 was its sheer size and con-

DR. DOBB'S JOURNAL **Reader Service Card**

Name ______________________ Phone ____________

Address ______________________________

City/State/Zip ______________________________

Expiration Date: Dec. 31, 1985 **September 1985 #107**

Please circle one for each category:

I. My firm or department is a:
- A. Systems Integrator/House
- B. Software Dev. Firm
- C. Hardware OEM or Manuf.
- D. DP, MIS or Data Service
- E. Consulting Firm
- F. Eng. or Science Lab.
- G. Other

II. My job function is:
- H. Company Mgmt./Admin.
- J. Computer Systems Mgt.
- K. Programmer/Technical Staff
- L. Consultant
- M. Engineering Mgmt. or Staff
- N. Scientific Mgmt. or Staff
- O. Other

III. Number of employees in my firm:
- 1. Less than 10
- 2. 10—99
- 3. 100—499
- 4. 500—9,999
- 5. 10,000—or More

IV. This inquiry is for:
- P. Immediate Purchase
- Q. Future Project

V. Purchasing Authority
(check all that apply)
- R. Recommend or Specify
- S. Final Decision-Maker
- T. No Influence

VI. I advise others about computers, on the average:
- 6. More Than Once-A-Day
- 7. Once-A-Day
- 8. Once-A-Week
- 9. Not At All

VII. I design/write software professionally
- U. Yes
- V. No

VIII. I buy computer products through:
(check all that apply)
- W. Retail Stores
- X. Mail Order Houses
- Y. On-site Direct Salespeople
- Z. All of the Above

IX. I am currently a subscriber:
- A. Yes
- B. No

Check each advertisement for corresponding number and circle below:

001	011	021	031	041	051	061
002	012	022	032	042	052	062
003	013	023	033	043	053	063
004	014	024	034	044	054	064
005	015	025	035	045	055	065
006	016	026	036	046	056	066
007	017	027	037	047	057	067
008	018	028	038	048	058	068
009	019	029	039	049	059	069
010	020	030	040	050	060	070
071	081	091	101	111	121	131
072	082	092	102	112	122	132
073	083	093	103	113	123	133
074	084	094	104	114	124	134
075	085	095	105	115	125	135
076	086	096	106	116	126	136
077	087	097	107	117	127	137
078	088	098	108	118	128	138
079	089	099	109	119	129	139
080	090	100	110	120	130	140

				Articles		
141	151	161	171			
142	152	162	172	181	191	201
143	153	163	173	182	192	202
144	154	164	174	183	193	203
145	155	165	175	184	194	204
146	156	166	176	185	195	205
147	157	167	177	186	196	206
148	158	168	178	187	197	207
149	159	169	179	188	198	208
150	160	170	180	189	199	209
				190	200	210

☐ **Please send me a one year subscription to Dr. Dobb's Journal at $25.00 and bill me later.**

Note:
For quicker, more effective processing of your inquiry, please provide responses to ALL requested information.

DR. DOBB'S JOURNAL **Reader Service Card**

Name ______________________ Phone ____________

Address ______________________________

City/State/Zip ______________________________

Expiration Date: Dec. 31, 1985 **September 1985 #107**

Please circle one for each category:

I. My firm or department is a:
- A. Systems Integrator/House
- B. Software Dev. Firm
- C. Hardware OEM or Manuf.
- D. DP, MIS or Data Service
- E. Consulting Firm
- F. Eng. or Science Lab.
- G. Other

II. My job function is:
- H. Company Mgmt./Admin.
- J. Computer Systems Mgt.
- K. Programmer/Technical Staff
- L. Consultant
- M. Engineering Mgmt. or Staff
- N. Scientific Mgmt. or Staff
- O. Other

III. Number of employees in my firm:
- 1. Less than 10
- 2. 10—99
- 3. 100—499
- 4. 500—9,999
- 5. 10,000—or More

IV. This inquiry is for:
- P. Immediate Purchase
- Q. Future Project

V. Purchasing Authority
(check all that apply)
- R. Recommend or Specify
- S. Final Decision-Maker
- T. No Influence

VI. I advise others about computers, on the average:
- 6. More Than Once-A-Day
- 7. Once-A-Day
- 8. Once-A-Week
- 9. Not At All

VII. I design/write software professionally
- U. Yes
- V. No

VIII. I buy computer products through:
(check all that apply)
- W. Retail Stores
- X. Mail Order Houses
- Y. On-site Direct Salespeople
- Z. All of the Above

IX. I am currently a subscriber:
- A. Yes
- B. No

Check each advertisement for corresponding number and circle below:

001	011	021	031	041	051	061
002	012	022	032	042	052	062
003	013	023	033	043	053	063
004	014	024	034	044	054	064
005	015	025	035	045	055	065
006	016	026	036	046	056	066
007	017	027	037	047	057	067
008	018	028	038	048	058	068
009	019	029	039	049	059	069
010	020	030	040	050	060	070
071	081	091	101	111	121	131
072	082	092	102	112	122	132
073	083	093	103	113	123	133
074	084	094	104	114	124	134
075	085	095	105	115	125	135
076	086	096	106	116	126	136
077	087	097	107	117	127	137
078	088	098	108	118	128	138
079	089	099	109	119	129	139
080	090	100	110	120	130	140

				Articles		
141	151	161	171			
142	152	162	172	181	191	201
143	153	163	173	182	192	202
144	154	164	174	183	193	203
145	155	165	175	184	194	204
146	156	166	176	185	195	205
147	157	167	177	186	196	206
148	158	168	178	187	197	207
149	159	169	179	188	198	208
150	160	170	180	189	199	209
				190	200	210

☐ **Please send me a one year subscription to Dr. Dobb's Journal at $25.00 and bill me later.**

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #27346, PHILADELPHIA, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal
P.O. BOX 13851
PHILADELPHIA, PA 19101

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #27346, PHILADELPHIA, PA.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal
P.O. BOX 13851
PHILADELPHIA, PA 19101

sequent inaccessibility. Most users of TeX78 would not readily have believed that the current, more powerful version of the program could possibly be made available in the small systems world. But here it is in two different versions for the PC-XT and its look-alikes. As we shall demonstrate below, it takes a large XT to run TeX, and it takes a good deal of disk space to store it, but both programs are genuine TeX, and have been tested and validated.

Throughout this review, we will refer to two TeX-related publications: *The TeXbook* and *TUGboat*. *The TeXbook* is the manual that describes TeX. Written by Donald Knuth and published by Addison-Wesley, it costs $14.95 in bookstores. The *TUGboat* is the newsletter of the TeX Users Group (also known as "TUG"). No matter which version of TeX you purchase, we believe that you will want to become a member of the TeX Users' Group. The initial yearly membership cost is $20.00, and this includes a subscription to the *TUGboat*. TUG is the best way to keep up-to-date with the latest TeX developments and the *TUGboat* is an excellent place for you to share your own TeX discoveries. Write to the TeX Users' Group, at P.O. Box 9506, Providence, Rhode Island 02940 for more information.

The distribution of MicroTeX and PCTeX marks a break from the pattern established under the sponsorship of the TeX Users Group for larger systems. The broad aim for larger systems through the network of TUG site coordinators has been to place full source code in the hands of the systems programmer at each location, so that the site will have full control over compilation and validation. This approach has depended on the assumption that the site will have a great deal of directly or indirectly addressable programming memory, and a robust and fully functional Pascal compiler. (In the past two years, experience has led to the formation of "David Fuchs' Law of Pascal Compilers," that TeX will smoke out a bug in every compiler created earlier than TeX itself.)

Even on a mainframe system, a TeX compilation can take hours (some have looked as if they were going to run for days), and the compilation time on a PC must be quite alarming. These two versions of TeX for the PC give the user the benefit of the special efforts of two long established members of the TeX community. Lance Carnes, the author of PCTeX, is the TUG site coordinator for "small" TeX. David Fuchs, the author of MicroTeX, is second only to Donald Knuth in his long association with TeX. Both versions produce an identical output file from identical input files (more later about the output format, called `DVI`), and both have gone successfully through the "torture test" validation prescribed by Donald Knuth for true implementations of TeX. The implementation methods, however, are quite different. MicroTeX was developed through a translation from Pascal to C, while PCTeX has followed the usual route of direct Pascal compilation. (Microsoft's Pascal compiler came through rather handsomely in this effort, with only one

bug.) There are various arcana complicating the above histories. David Fuchs wrote a post-optimizer for the C compiler to improve performance, and Lance Carnes had to work up a collection of segmented pointer types for Pascal along with the address calculation routines to go with them.

Each version of TEX is delivered in a large box. MicroTEX arrives with a copy of *The TEXbook* (also available separately in bookstores), a 92 page installation and introduction manual and 8 diskettes. The diskettes contain TEX, fonts, and DVI-EPS, a program that prints TEX's output files on the IBM graphics printer and its near relatives. We tested MicroTEX on an XT. We understand that the current version also runs on an AT.

PCTEX's box includes a binder containing a 26 page installation manual, a 128 page introduction to PCTEX, including a description of their own macro package, intended for beginners, a summary of the AMS-TEX macro package, and a reprint of the manual for the LATEX macro package. A program to print output on IBM and Epson graphics printers and *The TEXbook* can be purchased separately. PCTEX distributes TEX, its associated font tables, and the macro packages on 5 diskettes. If PC-DOT, the program to print the output, is purchased, an additional 8 diskettes contain the program along with its fonts. Two versions of TEX are provided—one for a PC with 512K-bytes of memory and another for one with 640K-bytes. MicroTEX automatically reconfigures itself to use as much memory as is available, from 512K-bytes to 640K-bytes. We successfully installed and tested PCTEX on both an XT and an AT.

The system requirements for these two versions of TEX are similar. Each requires an IBM PC with a hard disk, a floppy disk drive, and at least 512K-bytes of RAM. You will need about 5 megabytes of hard-disk storage although this amount can be trimmed down once the system is installed.

Taking the MicroTEX distribution first, it gets very high marks for ease of installation. At first accidentally, but then deliberately, I made every non-destructive error I could think of, and MicroTEX was forgiving of all of them. There is one slight warning I would give, however. After TEX is loaded, you are instructed to copy in the compressed version of the fonts, and then to get the process of decompression started and walk away for the 50 or so minutes it takes. Not quite. Immediately after the process starts, you have to supply the disk containing the system's `COMMAND.COM` file in drive `A`. If you have walked away too early, you will come back to find that nothing at all has happened in the past hour.

Other than that, everything is smooth. Within two hours, TEX is running, and you can try out the sample files on the directory, one of which is called `SAMPLE.TEX`. MicroTEX offers four levels of print quality on the Epson printer, and the observed times for each are given below. (`SAMPLE.DVI` is a reasonable test, since it loads a fair number of fonts.) At this point, if you are running the minimum 512K-byte configuration on your XT, you may run into trouble, however. Print your `SAMPLE.LOG` file, and DOS will load the printer driver into memory. Now try to run TEX, and you will be told, "Not enough memory for TeX." The MicroTEX manual warns the user that it may be necessary to leave out some of the resident drivers, but with DOS 3.0 in 512K-bytes, it is necessary to leave them all out. One might almost imagine that the program was distributed as an incitement to purchase that last 128K-bytes of memory.

PCTEX also installs smoothly, although we didn't deliberately try to make errors this time. The installation manual provides a step by step description, and installing TEX takes about half an hour. Installing PC-DOT and its associated fonts takes another hour or so—as with the MicroTEX installation, most of this time is spent waiting for fonts to be uncompressed. Unlike the MicroTEX installation, you have to attend to PCTEX during the installation process, swapping floppies as requested.

The installation of PCTEX involves a few more steps than the installation of MicroTEX, but you gain certain advantages as a result. To understand these it is necessary to consider how TEX is put together to make a working typesetting system. A pristine version of TEX is a very minimal typesetting program indeed. It provides the necessary primitive operations for character positioning at the lowest possible level, a sort of "assembler code" for typesetting. No reasonable person would even consider formatting a page at this level. The other side of the program is an extremely powerful macro processor. (The macro processor was an afterthought and an adjunct to the old TEX78, but it is at the very heart of the present program.) Only after a large number of macros has been processed and evaluated do you have a fully operational typesetting system. All currently available versions of TEX come with a macro file known as `PLAIN.TEX`, which forms the basis for any further refinements. `PLAIN.TEX` provides a great variety of convenient handles for common typesetting conventions, preloads a number of fonts, and arranges for the input of the hyphenation dictionary that is one of TEX's most remarkable features. (The stories of incompetent hyphenation algorithms in computer-assisted typesetting systems make up quite a tale of horror.) Getting all this stuff converted to tokens and premasticated into an efficient form for TEX to use takes time and the large-machine distributions of TEX actually provide for two separate compilations of the program, one known as `INITEX` to do the massaging and cough up a predigested format file, and `VIRTEX`, a second, smaller version with a weaker digestive system, which can read in the predigested `PLAIN.FMT` (notice the change in suffix) at a relatively high speed. In MicroTEX this has all been done for you, and the preloaded TEX is instantly ready to run. In PCTEX you get to do

it yourself.

The executable file provided with PCTeX is a VIRTEX with a special hook provided to call in the capabilities of INITEX as an overlay. The initial run of TeX is done with a special flag on the command line, and is devoted entirely to masticating and regurgitating the PLAIN.TEX file. (These alimentary metaphors are extremely pervasive in the documentation of TeX.) From then on, although you will be told that the TeX you are running has no format preloaded, the program will read in the PLAIN.FMT file automatically, without being prompted, and at more or less the optimum speed for a disk to memory transfer. As a result, PCTeX takes about 20 seconds longer to go into action than MicroTeX, but it gains in many important ways. One of these is immediately evident from listing the files in the TEXINPUT directory. As we noted above, PCTeX comes with both AMS-TeX for advanced mathematics formatting and with LaTeX, which is a powerful document formatting package using a syntax that allows you to describe a document by its logical structure instead of by its appearance. Both of these macro files are large—LaTeX is immense, requiring 640K-bytes to run—and you would soon weary of waiting for the macro-interpreter to do its thing with them if you had to read them in at the start of every invocation of TeX. The LaTeX or the AMS-TeX format file, however, can be read into place in almost the same time as the smaller PLAIN.FMT.

Another, even more subtle advantage of having an INITEX available has to do with the hyphenation table. Hyphenation rules differ considerably from language to language, and a user with a large amount of French or German text to set will more or less be forced to create a new FMT file to get satisfactory results. The creation of a new hyphenation table is non-trivial, and neither version of TeX for the PC provides the program that does it, but assuming that PATGEN has been run on some larger system, it still remains that the only practical way to get at the new hyphenation patterns is through the use of INITEX. Incidentally, this question of hyphenations is one of the only genuine limitations in the use of TeX for continuous text. There is not, at present, any way of switching from one hyphenation pattern to another in the course of a run. You may hyphenate efficiently in English, or in French, but not in both together.

Once either MicroTeX or PCTeX has begun reading the user input file there is very little difference between them. The TeX TRIP.TEX "torture-test" guarantees that we can assume that they are functionally identical and that they correctly recognize and process TeX. It is theoretically possible that a TeX might pass the torture-test and later produce a DVI file that was subtly different from that produced by all other TeXs, but it has not happened yet, and if it does, the test will probably be revised to eliminate the error. The normal benchmark is the setting of the TeXbook, and in announcements in a recent edition of *TUGboat*, the XT speed for this effort was given as 26.4 seconds a page for MicroTeX and "about 25 seconds a page" for PCTeX. In normal use, that is a very close race indeed. (PCTeX's running time on the AT is about six seconds a page.)

A true TeX-hacker will notice another difference between the two programs. MicroTeX is based on the latest version of TeX (we now have Version 1.4), while PCTeX is based on a somewhat older version (1.0, in our case). Usually, this would not be significant since the change from version to version has been quite small. Unfortunately, however, Version 1.3 incorporated major changes to TeX's memory management routines that greatly reduced the possibility of running out of memory while TeX-ing a document. We are assured that PCTeX will be brought up to date soon.

The version of LaTeX distributed with PCTeX is also outdated, which may cause some difficulties if you

try to move those files to another computer since LaTeX is still being developed and is undergoing rapid change. We transferred over the latest version of LaTeX and were able to run it on PCTeX after shortening one font name that was too long. If you are running a newer version of LaTeX on another computer, we expect that you will want to update PCTeX's copy before building the `FMT` file.

We understand that $\mathcal{A}\mathcal{M}\mathcal{S}$-TeX is now being developed using PCTeX, so we believe that the version distributed here is completely up to date.

Throughout this review, we have mentioned TeX's device-independent output file. Now, the time has come to turn back to general issues and to consider the implications of this.

In order to appreciate the nature of the `DVI` file, it is helpful to know something about the actual genesis of TeX. TeX is not a 'word-processor' in any ordinary sense of that mildly repellent term. The target machine for the original program was a genuine digital phototypesetter, using silver halide typesetting film to record characters painted on a CRT. That particular machine, in fact, has the highest resolution known in the industry—almost four times that of its nearest rival. But TeX does not stop even there. It produces a file for an imaginary typesetting device with a resolution of $1/2^{16}$ printer's points (there are 72.27 points to the inch), and leaves it to a second program to interpret that file for real-world devices. This output file is known as the 'device-independent' (`DVI`) file and is guaranteed to be exactly the same for any given source file when produced by a properly installed and tested TeX.

This notion of device independence goes far deeper than that in most other systems. It allows such an absolute confidence in the quality of results that the `DVI` file has seriously been proposed as a standard for document interchange in Europe. The suggestion is that on-line document retrieval from libraries be sent in `DVI` format for selective printing at the requesting site. But this is not to claim that TeX and the `DVI` file are the preferred system for all applications. The principal virtues of TeX carry with them certain constraints.

Perhaps the least generally understood are the constraints inherent in writing for a high-resolution phototypesetting system. At the present time, and with the presently available fonts, TeX optimizes for the best output device available. On an Alphatype, Autologic, Compugraphic or any of the other high-resolution typesetters for which `DVI` interpreters have been written, the interword spacing and inter-letter spacing are evaluated with round-off errors too small for the human eye to detect. At the most popular laser-printer and electrostatic resolutions (200, 240, and 300 dots to the inch) round-off errors are usu-

ally inoffensive, but they are nearly always discernable. It is simply impossible to do justice to the delicacy of high-resolution typesetting at 300 dots to the inch. (When we say that a device's resolution is, for example, 200 dots to the inch, we mean that the individual dots making up an image can be positioned to a resolution of 1/200*th* of an inch.)

This is not to say that the results are bad. Several thousand happy users of TeX have never seen their output at any resolution better than 200 dots to the inch, but others have wondered why the program cannot be better tuned to laser-printer resolutions and to the newer carefully designed laser-printer fonts such as Bigelow and Holmes's `Lucida`. The answer is that it can be so tuned, through the use of the 'TeX Font Metric' `TFM` file, a file which provides TeX with all the vital statistics about font characters, such as height, depth, width, letter-spacing adjustments (kerning), etc. If the manufacturer or distributor is willing to provide the necessary information, a `TFM` file can be made up to match any font currently available. It is not even a very laborious job. But a `TFM` file is not a font. It will not produce characters on any output device at all.

In some senses this can be an advantage. Suppose you wish to produce a book in the Baskerville font, and you have found a typesetter manufacturer who will provide you with sufficient information about its Baskerville fonts to produce the needed `TFM` file. For your final, high quality output, you have the problem solved; TeX will read the `TFM` file and make all its calculations based on the character widths, etc., of that font. But you do not want to do the proof copies on a slow, expensive photographic processor, and there is no 300-dot/inch Baskerville on your laser printer, far less on your graphics impact printer. If you are willing to shut your eyes to the bad letter spacing, and are concerned primarily with correcting typos and misspellings, you can make some font that you do have on your low to medium resolution printer masquerade as Baskerville. The laser printer output will look dreadful, but the cost of proof correction will be kept within reason. Conversely if the laser printer output is what you really want, there is no reason to use an ersatz high-resolution font at all. Create a `TFM` file that genuinely matches one of the fonts specially designed for use at 300 dots/inch and your laser-printer output will be very acceptable indeed.

It is even possible to create a `TFM` file and an associated format file for a daisy wheel printer. To run TeX in this mode is to give up at least nine-

tenths of its power as a typesetting system, but it can be done. The important thing is to recognize that letter-spacing and interword spacing are governed by the values supplied in the TFM file, and once the choice has been made, the DVI format will record that choice with breathtaking accuracy. No matter what the resolution is of your target machine, the DVI file will represent something far better than the best that that machine is capable of.

So while there are no technical reasons why TeX output cannot be tuned to particular output devices, such tuning is almost never done in practice, and in fact this tuning is usually considered to be an undesirable thing to do. The reason for this is that tuning introduces device and resolution dependencies into the DVI file and these dependencies will negate the more general concept of device independence provided by TeX. For not only are the positionings of the characters of a page described in a device-independent manner, but the characters themselves are also defined in a device-independent fashion.

The fonts used by TeX are produced from descriptions—programs, if you will—written in a language called **METAFONT**. A **METAFONT** program defines a font's characters by specifying the movement of a pen over a Cartesian coordinate system. As the name of the system implies, a "meta-font" is defined in this fashion—in other words, a single **METAFONT** description can be used to produce a *family* of related fonts (for example, separate fonts in roman and bold face) by varying the parameters that control values such as the pen shape and size. The important result in the context of this review, is that the description also is resolution-independent. In other words, the description will produce font bitmaps for use at different resolutions when the appropriate parameters are altered.

At present, only one generally available family of fonts, the Computer Modern family, has been defined through **METAFONT**. Variations within the family provide serifed roman, bold, italic, and slanted faces, a typewriter face, and a set of sans-serifed faces, each in a range of point sizes. TeX use is tightly coupled to the Computer Modern font family because of the flexibility afforded by the **METAFONT** description.

Since the Computer Modern family of fonts can be generated at almost any resolution needed by a device, and since the DVI file format represents the output with a precision unattainable by any device, the TeX concept of "device independence" can be restated as providing the *same* output on *every* output device, constrained only by the limitations of the device. There are two aspects to this formulation of device independence. One is that of *source-level independence*. In other words, every implementation of TeX should produce an identical DVI file from a TeX source file, if that source file uses the standard TeX environment—the Computer Modern fonts and the facilities defined in *The TeXbook*. The second aspect is that of *output-level independence*. A DVI file produced by any implementation of TeX should be able to be printed on any suitable printer, if the standard TeX environment has been used. DVI files can be, and frequently are, transferred from one model of computer to another in order to take advantage of the printers attached to the particular computers. As a practical aside, the most frequently encountered problems in developing the mechanisms that allow the transferring of a DVI file from one computer environment to another are related to the differing computer architectures and to the file transfer itself. The DVI file is defined as a stream of 8-bit bytes. Finding a file transfer mechanism that allows transmission of all eight bits of the byte sometimes involves a time consuming search. Differences in the ways in which these bytes are packed into the computer's words also may cause difficulties that require careful attention to overcome.

As the standard TeX environment is so important to maintaining device independence, you should consider the capabilities and cost of the printer that you will need carefully before deciding to purchase TeX. To get the full benefit of TeX, the output device you choose must be powerful enough to simulate TeX's ideal device and further, your device must allow you enough flexibility to allow you to use the TeX fonts.

What this means is that you should not plan to use a mechanized typewriter with TeX. Technically, you certainly could build the TFM font width tables to describe the typewriter's characters, but practically, TeX documents produced in this way will be incompatible with everyone else's and you will be unable to use most of TeX's more powerful features. What you will want to have is either a printer that allows you to do full page bit graphics or one that allows you to define your own characters and place them arbitrarily on the page (or, of course, one that allows you to do both).

Very few printers can print a DVI file directly. Generally, the DVI file is converted into a form compatible with the printer by an application program called, in TeX lingo, the *Output Driver*. As a practical matter, before you buy a printer for TeX use, you should make sure that the appropriate Output Driver software already exists for your computer. Writing such a conversion program is not an extraordinarily hard programming project, but getting the output to look "right" involves surprising subtleties. It is often more convenient to purchase the conversion software so you can get directly to the matter of creating beautiful documents.

Printers appropriate for TeX use can be placed into three different categories, based on the resolution at which fonts can be defined and at which elements of the image can be placed.

Low-resolution printers are usually impact printers. The lowest resolution at which recognizable output can be produced is about 75 dots/inch. Some printers of this type can print at resolutions up to 200 dots/inch. At the lowest resolutions, the output from these printers

is often unreadable, particularly in mathematical equations and in footnotes where the type sizes tend to become quite small. At the highest resolutions, the printers can take quite long to print each page—up to five minutes per page in some cases. Both PCTeX and MicroTeX make available software to print TeX output on a number of these printers (both products support the IBM Graphics printer and the Epson RX and FX printers).

At the other end of the range are *high-resolution printers.* These print at resolutions greater than 1000 dots/inch and are generally phototypesetters—devices producing output on photographic paper that must be developed before the image can be inspected. They are frequently quite slow (several minutes per page of output) and quite expensive to purchase and to operate. They are, however, the only acceptable way to produce output suitable for publication in high-quality books. Fortunately, a number of businesses own phototypesetters and are equipped to print TeX `DVI` files from IBM PC disks. The advertisements in the TeX Users Group Newsletter, the *TUGboat,* are the best way to locate these businesses.

In the middle is an increasingly important group of devices: the *medium-resolution printers.* Most of these printers are *laser printers,* and the 300 dot/inch resolution is becoming somewhat of a standard. While it is not possible for devices of this resolution to produce true book publication-quality output (the advertising claims of manufacturers notwithstanding), it is quite realistic to expect to be able to use these printers to produce output for less demanding applications such as newsletters, memos, and proofs of material that will ultimately be sent out for typesetting on a high-resolution printer. Some of the laser printers are not suitable for TeX use, as will be discussed, but those that have successfully printed TeX output range in price from about $4000 upward and print at speeds from about five pages per minute upward.

A laser printer is built from a *marking engine* and a *controller.* The marking engine is what puts the marks onto the paper. The controller drives the marking engine and provides the hardware and software interfaces that are visible to the person programming an application to print on the laser printer.

When comparing different laser printers, it is important to distinguish between the functions of marking engine and those of the controller. For example, it is usually correct to assume that maintenance statistics for one laser printer will also hold for a different printer if the two use the same marking engine. It is incorrect, however, to assume that information on the graphical capabilities of one manufacturer's laser printer will be valid for another's, even if the two printers both use the same marking engine.

As illustration, one commonly used marking engine is the Canon LBP-CX. This marking engine, used by many different laser printer manufacturers, is the heart of the Imagen 8/300, the QMS 800, the

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

PC - DOT Draft

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

DVI - EPS Level 1

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

DVI - EPS Level4

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

Apple Laser Writer

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

PC - DOT Final

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

DVI - EPS Level 2

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

DVI - EPS Level 3

Perhaps an example taken from the TEXbook will illustrate this best. This is exercise 18.40 from the chapter *"Fine Points of Mathematics Typing"*.

$$\sum_{p\ \mathrm{prime}} f(p) = \int_{t>1} f(t)\, d\pi(t).$$

The DVI output from this paragraph is shown in various resolutions from the lowest MicroTEX resolution on the printer up to the 5333 line/inch resolution of the Alphatype CRS.

Alphatype CRS

Figure

Samples of TEX Output

HP LaserJet, and the Apple LaserWriter. What distinguishes these laser printers from one another are the interfaces implemented by their controllers. The HP LaserJet, for example, is essentially an upgrade for a mechanized typewriter-like device. It does not allow fonts to be loaded over its hardware interface line and full-resolution bitmaps are limited to about one quarter page. We do not know of software that allows printing of TEX's output format on the HP LaserJet and suspect that such software will be extremely difficult, if not impossible, to write because of the printer's limitations. On the other hand, the Apple LaserWriter implements an interface language called PostScript. PostScript is a general purpose programming language, Forth-like in appearance, and is one of the most sophisticated printing languages available (PostScript is based on an earlier Xerox-defined printing language, which is named InterPress). The LaserJet and the LaserWriter use the same marking engine but are radically different in the sophistication of the functions that they provide.

Indeed, to a programmer providing TEX support, the interface language defined by the laser printer's controller is usually more important than which marking engine is being used. The interface language is generally the same or similar for a company's laser printers, even though those products may use significantly differing marking engines.

The companies that make PCTEX and MicroTEX have announced that they will be supporting TEX output from the IBM PC for the Apple LaserWriter, the Imagen laser printers, and the QMS laser printers. Another relatively low cost laser printer, the DEC LN-03, has been supported, although not yet on the IBM PC. A full list of supported devices is printed in each edition of the *TUGboat*.

An additional difference between the two implementations becomes apparent when comparing the two Epson drivers. DVI-EPS, included with the MicroTEX package, pro-

vides four levels of printing on the Epson. We printed a fairly dense page of TeX output at each of these levels (see Figure, page 88) and observed that the time to print the page ranged from about $22\frac{1}{2}$ minutes at the best quality (level 1) to only $3\frac{1}{2}$ minutes at the lowest quality (level 4). Intermediate timings were 12 minutes for level 2 and 10 minutes for level 3. In contrast, PC-DOT, available as an option with PCTeX, only provides two printing levels. The draft mode produced output similar to DVI-EPS's level 2, and took $12\frac{1}{2}$ minutes for our sample page. Final mode in PCTeX is comparable to DVI-EPS' level 1, and took $23\frac{1}{2}$ minutes to print. In all cases, print time seemed to depend more on the density of the page than on other factors.

23 minutes is a substantial time to wait for a page of output to be printed. Consequently, we believe that DVI-EPS is significantly preferable to PC-DOT. In particular, DVI-EPS' lowest quality level 4 printing at under four minutes for the page will get quite heavy use during document development and proofing. What is desperately needed are mechanisms that allow the `DVI` file to be previewed on the PC's screen, allowing the slow printing pass to be bypassed altogether.

In summary, despite the essential similarity of output from the two implementations of TeX for the PC, there are numerous differences that allow for a meaningful comparison. MicroTeX comes as a complete package, including TeX, the format file, the DVI-EPS printer driver and its numerous fonts at several different pixel-densities, and the authoritative documentation of *The TeXbook* itself. The version of TeX is absolutely up to date, and the driver offers a wide range of output quality. All of this together comes to \$495.00. You can run the program in 512K-bytes of memory, but only at the price of some inconvenience (no resident drivers in your DOS).

PCTeX offers various options. The basic price of \$279.00 gives you TeX, separately compiled for two different sizes of memory, and the AMS-TeX, and LaTeX macro files, together with the mechanism for loading them efficiently. At the moment PCTeX is still at version 1.0 by contrast with MicroTeX at version 1.4. The PC-DOT driver program at \$100.00 is not so broadly functional as the DVI-EPS program, If you want *The TeXbook*, you must order it separately for \$15, but you do get the user documentation for LaTeX and AMS-TeX.

In our judgment, the ideal package would be PCTeX for the TeX together with DVI-EPS to drive the printer. Only those with larger pockets might consider this, since it requires buying both PCTeX and MicroTeX. Otherwise, especially for those with technical documentation in mind, we strongly favor the PCTeX offering. We believe that the inclusion of `INITEX` capabilities in PCTeX is so important that it overrides the inconveniences caused by the lag in versions and by the less functional printer driver.

(Addison-Wesley is distributing a complete `INITEX` *in the August update of MicroTeX.-Ed.)*

[This review was formatted with MicroTeX on an IBM AT with 512K in $2\frac{1}{2}$ minutes. The fully-composed output was uploaded to a Stanford mainframe and sent to an Autologic APS-micro-5 typesetter, with the results seen here.]

PCTeX, Version 1.0
Company: Personal TEX, Inc.
20 Sunnyside, Suite H
Mill Valley, CA 94941
(415) 388-8853
Price: \$279.00; PCDOT \$100.00

MicroTeX, Version 1.4A1
Company: Addison-Wesley Publishing Company, Inc.
Reading, MA 01867
(617) 944-3700
Price: \$495.00

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197**.

Dr. Dobb's C Tools On Disk

To complement **The Toolbook** Dr. Dobb's also offers the following programs on disk for only $19.95 each. Full source code is included and, except where indicated, both CP/M and MS or PC DOS versions are available.

Small-C Compiler

Jim Hendrix's **Small-C Compiler** is the most popular piece of software published in Dr. Dobb's 10-year history. Like a home study course in compiler design, the **Small-C Compiler** and **The Small-C Handbook** provide all you need to learn how compilers are constructed, as well as teaching the C language at its most fundamental level. **The Small-C Handbook** provides documentation for both versions ; however, an addendum is recommended in addition to **The Handbook** for MS or PC DOS-specific documentation. The addendum is available for $4.95.

Small Tools: Programs for Text Processing

This package consists of programs designed to perform specific functions on text files, including: editing; formatting; sorting; merging; listing; printing; searching; changing; transliterating; copying and concatenating; encrypting and decrypting; replacing spaces with tabs and tabs with spaces; counting characters, words, or lines; and selecting printer fonts.

Documentation available for $9.95.

Small-Mac: An Assembler for Small-C

Small-Mac is a macro assembler designed to stress simplicity, portability, adaptability, and educational value. The package features simplified macro facility, C-language expression operators, descriptive error messages, object file visibility, and an externally defined machine instruction table. Included programs are: macro assembler, linkage editor, load-and-go loader, library manager, CPU configuration utility, and dump relocatable files. This program is available for CP/M systems only.

Documentation available for $9.95.

To order, send this order form with your payment to: **Dr. Dobb's Journal, 2464 Embarcadero Way, Palo Alto, CA 94303**

	CP/M	MS or PC DOS	Price		QTY.		TOTAL
BOOKS							
Dr. Dobb's Toolbook			$29.95	X		=	
Small-C Handbook			$17.95	X		=	
DISKS (Check Format)							
Small-C Compiler			$19.95	X		=	
Small Tools Text Processor			$19.95	X		=	
Small Mac Assembler (for CP/M only)			$19.95	X		=	
MANUALS							
Small-C Compiler MSPC-DOS specific addendum to The Small-C Handbook (NOTE: The Small-C Handbook provides full documentation for the CP/M version)			$4.95	X		=	
Small Tools			$9.95	X		=	
Small Mac			$9.95	X		=	

Sub Total $

California residents add applicable sales tax — Tax

Add $1.75 per item for shipping in U.S., $4.25 per item outside U.S. — Shipping

TOTAL $

For CP/M system disks only, please specify one of the following formats:

- ☐ Kaypro
- ☐ Apple
- ☐ Zenith Z-00 DS/DD
- ☐ Osborne
- ☐ 8″ SS/SD
- ☐ Inquire about other formats

PAYMENT MUST ACCOMPANY YOUR ORDER

☐ Check enclosed

☐ Please charge my ☐ VISA ☐ M/C ☐ Amer. Exp.

Card #________________

Exp. Date________________

Signature________________

Name________________

Address________________
(please use street address)

City________________

State________ Zip________

Allow 6–12 Weeks for delivery

3107

SCSI Your Mac

by John Bass

I got tired of waiting on those sluggish Mac floppy drives, tired of waiting for an affordable hard disk for the Mac. I got so tired, finally, that I went to work. My efforts paid off, and I was pleased enough with the result to think that others might appreciate my MacSCSI® interface, too.

One of the most obvious advantages to doing your own hard disk interface for the Mac is cost. I think MacSCSI is the hard-disk equivalent of *DDJ*'s January 1985 "Fatten Your Mac" 512K upgrade: the hacker's hard disk upgrade at budget prices, if you're willing to shop around for disks and SASI/SCSI controllers. Here in Silicon Valley we see hard disks at the computer swaps for between $100.00 and $500.00, often either refurbished or new discontinued models that were surplus. Likewise, SASI/SCSI controllers are seen at between $50 and $150. Thus, with the MacSCSI interface and some bargain hunting you can put a hard disk on your Mac for less than $500. New production drives and controllers are available between $900 and $3000, depending on the size and performance.

Another advantage is the experience you'll gain, both in writing your own bells and whistles into the supporting software, and in simply cracking open your Mac and getting to know its innards better. And in this regard, the MacSCSI interface is unlike the 512K upgrade: you don't have to take a soldering iron to a delicate, expensive logic board to install MacSCSI. You don't have to modify the Mac logic board at all.

The parts for this venture are few. The MacSCSI host adapter consists of four parts: an NCR5380 single chip SCSI interface, two 74LS10's to provide the address decodes, and a 50 pin header for the SASI/SCSI cable. The rest of a complete setup is a SASI/SCSI controller, hard disk drive, power supply, and cables for power and data. You can make your own case from a bookcase speaker for a few dollars, you can build a nice custom cabinet, or you can buy an off-the-shelf hard disk enclosure complete with power supply and cables.

Figure 1 (page 95) tells the story: it contains the schematic for the MacSCSI interface. The host adapter is memory mapped into the Mac's ROM address space above its last valid address. To accomplish this, I used selection equations for the NCR5380 of

$$NCRSEL = \overline{ROMSEL} * A20 * \overline{A11}$$
$$NCRRD = NCRSEL * A4$$
$$NCRWRITE = \overline{UDS}$$

By using address lines to enable the read and write enables on the NCR5380, I avoided needing to run any wires to chips or to cut away at the main circuit board; the only interface point to the Mac is its ROM socket. This scheme does, though, require that the software read an NCR5380 register at one address and write it at another—quite a reasonable tradeoff for not having to do any cuts or adds to the main logic board, I think.

While it's not clear whether Apple would honor a warranty or Apple Care contract on a Mac upgraded this way, if done carefully the upgrade will not physically alter or harm the Mac. If you have problems with your Mac (other than the MacSCSI upgrade), most helpful dealers should be willing to service your Mac if your boards are not altered or damaged. You may even find some helpful dealers and independent repair centers that will install the upgrade for you at their normal shop rates (about an hour of their time).

If you're up for a project, the interface can be assembled using off-the-shelf parts and point-to-point wiring. If you do it this way, you'll need to make a chip carrier from perf board with holes on 0.1 inch centers. You can salvage the socket pins for the chip carrier by cutting apart two Augat low profile 28 pin sockets with machined pins. Then attach the perf board to a wooden support block and drill out to 0.055-inch in the pin pattern for the Mac's ROMs. Keeping the wooden block attached, press the pins into the perf board using a vise or press (be careful not to crush the pins). The wooden block provides a die to protect the socket pins during the insertion process. You can then insert the other parts into the perf board and wire them using point-to-point soldering on the back side. Finally, and carefully, remove the ROMs from the board, insert them into the chip carrier, and plug the chip carrier with the ROMs back into the ROM sockets.

I'm skipping any discussion on how to take your Mac apart and reassemble it; for that, see the installation section on page 96.

Now for the hard part. I've come up with several options for getting the SCSI ribbon cable out of the case, none of them elegant. One is to cut a hole in the back of the case for an exit. Another is a panel-mounted connector installed in the case (my preferred solution). Or you can run a ribbon cable out between the bottom case seam in front, although you'll have to widen the seam slightly to do this. A fourth solution is not to bring the cable out at all, of course. General Computer has already demonstrated the feasibility of installing a hard disk inside the Mac case. I haven't

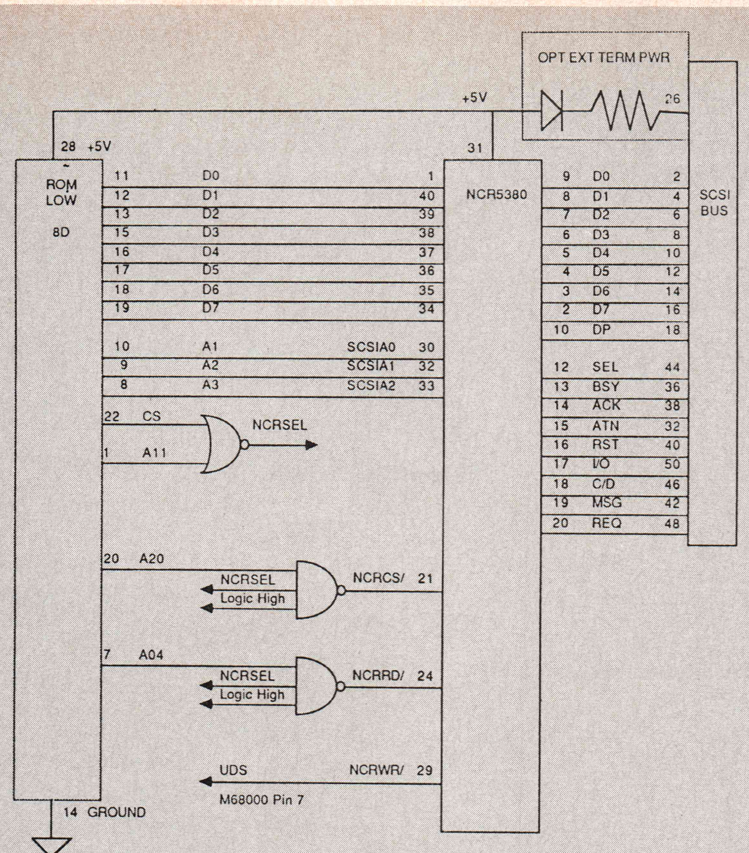


Figure 1

Mac SCSI host adaptor by DMS Design—Schematic Logic Drawing

Mac SCSI Selects

READ	WRITE	
RESET	INITIATOR START	0xE
DATA	TARGET START	0xC
STATUS	DMA START	0xA
SCSI STATUS	SELECT	0x8
TARGET CMD	TARGET CMD	0x6
MODE	MODE	0x4
INITIATOR CMD	INITIATOR CMD	0x2
SCSI DATA	SCSI DATA	0x0
0x50001X Odd Bytes	0x50002X Even Bytes	

Macintosh Memory Map

Bank	Address	Overlay	Overlay/
0	00xxxxx	Rom	Ram
1			
2	400xxxx	Rom	Rom
	50000xx	Mac SCSI	Mac SCSI
3	60xxxxx	Ram	
4	9FFFFx	SCC Read	SCC Read
5	BFFFFx	SCC Write	SCC Write
6	DFxxFF	IWM	IWM
7		VIA	VIA

Figure 2

Device Memory Maps

Figure 3

Template for Cut Out

Installation Outline for External Cabling

Materials Required

- Xcelite XTD-15 screwdriver with a 6-inch extension
- wooden ruler
- two cotton towels
- hand drill, ¼- and ⅛-inch drill bits
- 10-inch Mill Bastard flat file
- IC extractor or a small screwdriver to remove the 28 pin ROMs
- AMP connector 1-499970-0; AMP Ground Plane 102793-4
- paper template (see Figure 3, page 95)

Opening the Case

Disconnect the Mac from all external cables, power, mouse, keyboard etc. Place the Mac facedown on a towel to prevent marring the plastic case. Remove the five case screws with the Xcelite XTD-15 Torx screwdriver; two are located under the handle, one underneath the battery case plate, and two at the bottom of the case. Use the long edge of a wooden ruler to pry the case apart. Applying pressure to the power and I/O connectors, while lifting on the case, should release it from the faceplate. Set the foil EMI sheild aside until reassembly.

Cutting the connector slot

Using the paper template, mark the slot to be cut into the back of the Macintosh. Remove most of the plastic from the slot with a handdrill, and square off the edges with the flat file. With the slot cut, recenter the template, mark and drill the ⅛-inch holes for the 50 pin feed-thru header. Bolt the header to the inside of the case, with pin 1 on the bottom.

Removing the Motherboard ROMs

Disconnect the diskdrive and analog board cables from the motherboard; remove the motherboard from the chassis and place it on the second towel. The ROMs are in the IC sockets marked ROM HI and ROM LO. Using a chip extractor or small screwdriver, carefully remove the chip marked ROM HI. Place the ROM in the socket marked ROM HI on the Mac SCSI board, and repeat the procedure for ROM LO. Double check the orientation of the ROMs by noting that the notch on each chip points away from the 50 pin header.

Installing the Mac SCSI board

Orient the Mac SCSI board, with the 50 pin header toward the I/O connectors on the motherboard, and simply plug it into the vacated ROM sockets. Reinsert the motherboard into the chassis, taking care not to damage the 50 pin header. Reconnect the diskdrive and analog board cables, followed by the large Mac SCSI interface cable.

Closing the Mac back up

Set the Macintosh facedown and position the EMI sheild over the I/O connectors. Replace the back, making sure each side is completely seated, and secure the five holding screws.

tried this yet.

The software for MacSCSI is so far pretty simple. It consists of a routine to format the drive and a disk driver. Details of the operation of these two routines will vary a little depending upon the controller manufacturer and drive type. The Listing in this article (page 98) is for a XEBEC S1410 controller with rev D ROMs and a Seagate ST506 5Mb drive, both of which are common on the used market in Silicon Valley. As a starting point I used the RAMdisk supplied with the Aztec C Compiler release. The strategy was to use the basic RAMdisk driver as a local cache and use a simple LRU algorithm for replacement. Using the Aztec C example for the explorer desk accessory, we recoded the driver into C, for a slight loss in performance. On a 128k Mac the cache size should be set between 1 and 10. On a 512k Mac you should use some number between 30 and 300, optimally.

Other development environments and C compilers may be used as well with minor changes in the coding and installation procedures. For Aztec C the documentation on the RAMdisk and explorer desk accessory provide all the magic incantations necessary to convert the source files into an installed driver.

Questions, anyone? How large a disk, you ask? It depends in part on the software; you can reasonably handle up to about five Mb of storage on the Mac without partitioning; at ten Mb the number of files gets unwieldy. Then you'll just need more sophisticated software. Do you have to have a Fat Mac to support MacSCSI? No, but while this hard disk upgrade can work with many applications on a 128k Mac, a 512k Mac is highly recommended.

What if you don't want to take the time to do it yourself? For the lazy, the MacSCSI board is available from Fastime, P.O. Box 12508, San Luis Obispo, CA 93406 for $150.00 assembled and tested, along with machine-readable copies of the sources listed here. You can also get a complete kit, including drives from 10Mb to 110Mb, tape backup, and extended software drivers with multiple soft partitions to support the larger drives. Drivers for disk sharing and the Appletalk file server are planned or under development. Write for more information.

John L. Bass
DMS Design
P.O. Box 1456
Cupertino, CA 95014

DDJ

(Listings begin on page 98)

DDJ BACK ISSUES

#73 Volume VII, Issue 11:
Wildcard UNIX Filenames—Tests for Pidgin—68000 Cross Assembler Listing, Part 2—Adding More BDOS Calls—The Perfect Hash—BASIC Memory Management—Benchmarks for CP/M-86 vs. MSDOS, and the 8087.

#78 Volume VIII, Issue 4:
RECLAIM Destroyed Directories—Binary Magic Numbers—8080 Fig-Forth Directory & File System—SAY'' Forth Votrax Driver—TRS-80 8080 to Z80 Translator— Basic Disk I/O, Part II.

#79 Volume VIII, Issue 5:
The Augusta Compiler—A Fast Circle Routine—Enhancing the C Screen Editor—Shifts and Rotations on the Z80—The SCB, TSX, and TXS Instructions of the 6502 and 6800—MS-DOS vs. CP/M-86—Controlling MBASIC—The Buffered Keyboard—IBM PC Character Set Linker—Flip Utility for the IBM PC.

#80 Volume VIII, Issue 6:
Fast Divisibility Algorithms—B-Tree ISAM Concepts—CP/M BDOS and BIOS Calls for C—Serial Expansion in Forth—Fast Matrix Operations in Forth, Part I—Yes, You Can Trace Through BDOS—Julian Dates for Microcomputers—8088 Addressing Modes—8088 Line Generator—CP/M Plus.

#81 Volume VIII, Issue 7:
The Augusta Compiler, continued—RED: A Better Screen Editor, Part I—Anatomy of a Digital Vector and Curve Generator—Fast Matrix Operations in Forth, Part II—The AGGHHHI Program—MBOOT Revisited—CP/M Plus Feedback—MS-DOS Rebuttal—68000 Tools—Sizing Memory on the IBM PC.

#82 Volume VIII, Issue 8:
Serial-to Parallel: A Flexible Utility Box—McWORDER: A Tiny Text Editor—And Still More Fifth Generation Computers—Specialist Symbols and I/O Benchmarks for CP/M Plus—CP/M Plus Memory Management—Zero Length File Test—PAUSEIF, QUITIF, and now SKIPIF—ACTxx Cross Assemblers.

#83 Volume VIII, Issue 9:
FORTH ISSUE: Forth and the Motorola 68000—Nondeterministic Control Words in Forth—A68000 Forth Assembler—GO in Forth—Precompiled Forth Modules—Signed Integer Division—Some Forth Coding Standards—The Forth Sort.

#84 Volume VIII, Issue 10:
Unix to CP/M Floppy Disk File Conversion—A Small-C Help Facility—Attaching a Winchester Hard Disk to the S-100 Bus—Using Epson Bit-Plot Graphics—8086/88 Function Macros—Auto Disk Format Selection—CP/M Plus Device Tables.

#85 Volume VIII, Issue 11:
A Kernel for the MC68000—A DML Parser—Towards a More Writable Forth Syntax—Simple Graphics for Printer—Floating-Point Benchmarks.

#86 Volume VIII, Issue 12:
Faster Circles for Apples—Cursor Control for Dumb Terminals—Dysan's Digital Diagnositic Diskette—Interfacing a Hard Disk Within a CP/M Environment—The New MS-DOS EXEC Function.

#87 Volume IX, Issue 1:
A structured Preprocessor for MBASIC—A Simple Window Package—Forth to PC-DOS Interface—Sorted Diskette Directory Listing for the IBM PC—Emulate WordStar on TOPS-20—More on optimizing compilers—The PIP mystery device contest.

#88 Volume IX, Issue 2:
Telecommunications Issue: Micro To Mainframe Connection—Communications Protocols—Unix to Unix Network Utilities—VPC: A Virtual Personal Computer for Networks—PABX an the Personal Computer—BASIC Language Telecommunications Programming—U.S. Robotics S-100 Card Modem.

#89 Volume IX, Issue 3:
RSA: A Public Key Cryptography System, Part I— Introduction to PL/C: Programming Language for Compilers—Program Design Using Pseudocode—More on Binary Magic Numbers—How fast is CP/M Plus?—CP/M 2.2 BIOS Function: SELDSK—The results of the Floating-Point benchmark.

#90 Volume IX, Issue 4:
Optimizing Strings in C—Expert Systems and the Weather—RSA: A Public Key Cryptography System, Part II—Several items on CP/M Plus, CP/M 2.2 Compatibility—BDOS Function 10: Vastly improved—More on MS-DOS EXEC Function—Low-Level Input-Output in C.

#91 Volume IX, Issue 5:
Introduction to Modula-2 for Pascal Programmers—Converting Fig-Forth to Forth-83—Sixth Generation Computers—A New Library for Small-C—Solutions to Quirks in dBASE II.

#92 Volume IX, Issue 6:
CP/M on the Commodore 64—dBASE II Programming Techniques—First Chinese Forth: A Double-Headed Approach—cc-A Driver for a Small-C Programming System—A New Library for Small-C (Part II)—Comments on Sixth Generation Computers—Review of Turbo Pascal.

#95 Volume IX, Issue 9:
Forth Special Issuel—File Maintenance in Forth—Forth and the Fast Fourier Transform—Computing with Streams—A Forth Native-Code Cross Compiler for the MC68000—The FVG Standard Floating-Point Extension—CP/M Plus: Interbank Memory Moves Without DMA—Ways to make C more powerful and flexible.

#96 Volume IX, Issue 10:
More dBASE II Programming Techniques—Simple Calculations with Complex Numbers—GREP.C: A Unix-like Generalized Regular Expression Parser—An optimization scheme for compilers, MSDOS 2.0 Filters, Sizing RAM under MSDOS, Two programming systems illustrating Runge-Kutta integration.

#97 Volume IX, Issue 11:
Adding Primitive I/O Functions to muLISP—Program Monitor Package: Using Interrupts to Instrument Applications—CP/M 2.2 Goes PUBlic—A Guide to Resources for the C Programmer—RESORT.

#98 Volume IX, Issue 12:
Varieties of Unix—Unix Device Drivers—A Unix Internals Bibliography—A File Browser Program—An Introduction to Parsing.

#100 Volume X, Issue 2:
Festschrift for Doctor Dobb—Fire in the Valley—Tiny Basic for the 68000—An Enhanced ADFGVX Cipher System—More dBASE Tips & Techniques.

#101 Volume X, Issue 3:
Programming in Logic—Tour of Prolog—Tax Advisor: A Prolog Program Analyzing Tax Issues.

#102 Volume X, Issue 4:
What We Can Learn about Human Factors Engineering from a Game-Playing Computer—Shortcut to SCISTAR: For Prowriter Users—fx80char: A Character Editor for the Epson FX 80—A Magic Mushroom for the Epson Alice—Let's Mouse Around.

#103 Volume X, Issue 5:
Using Decision Variables in Graphics Primitives—Solid Shape Drawing on the Commodore 64—A Compiler Written in Prolog.

#104 Volume X, Issue 6
Information Age Issues; Modems: 2400 Bit/Sec and Beyond; C UART Controller; Christensen Protocols in C.

#105 Volume X, Issue 7
Build a Custom PC or Clone; The Ultimate Parallel Print Spooler; Designing a Real-Time Clock for the S-100 Bus.

#106 Volume X, Issue 8
C Compilers for MSDOS; A Peephole Optimizer for Assembly Language Source Code; Small C Update; Asynchronous Protocols.

TO ORDER:

Return this coupon with payment to: **Dr. Dobb's Journal**, 2464 Embarcadero Way, Palo Alto, CA 94303

Please send me the issue(s) circled: **71 72 73 78 79 80**
81 82 83 84 85 86 87 88 89 90 91 92 95
96 97 98 99 100 101 102 103 104 105 106

Price: 1 issue—$5.00, 2-5 issues—$4.50 each, 6 or more—$4.00 each.
(There is a $10.00 minimum for charge orders.)

I enclose $ __________ (U.S. check or money order).
Outside the U.S., add $.50 per issue.

Please charge my: ☐ Visa ☐ M/C ☐ Amer. Exp.

Card No. ______________________ Exp. Date ______

Signature ______________________

I have read the postal instructions and understand that I will not receive my order unless I have sent the correct payment amount.

Name ______________________

Address ______________________

City ______________________

State ______________ Zip ______________

Outside U.S., add $.50 per issue. Price includes shipment by second class or foreign surface mail. Within U.S., allow 9 weeks for delivery. For U.P.S. shipment, add $1.00 for 1-2 issues and $.50 per issue thereafter—NO P.O. BOXES. Airmail rates: Canada—add $1.75 per issue; other foreign add $3.00 per issue.

3107

```
all: clean FormatSCSI ChkSCSI MountSCSI MacSCSI BootSCSI
	echo done

clean:
	rm FormatSCSI MountSCSI MacSCSI BootSCSI ChkSCSI
	rm TestSCSI MSCSI
	rm *.bak

print:
	cat makefile > .bout
	cat newpage > .bout
	cat s1410.c > .bout
	cat newpage > .bout
	cat np_fmt.c > .bout
	cat newpage > .bout
	cat np_dvr.c > .bout
	cat newpage > .bout
	cat newpage > .bout

FormatSCSI:
	cc np_fmt.c
	cc s1410.c
	ln -mo FormatSCSI np_fmt.o s1410.o sys:lib/mixcroot.o -lc
	cprsrc DRVR 30 sys:system FormatSCSI
	rm np_fmt.o s1410.o

ChkSCSI:
	cc np_chk.c
	cc s1410.c
	ln -mo ChkSCSI np_chk.o s1410.o sys:lib/mixcroot.o -lc
	cprsrc DRVR 30 sys:system ChkSCSI
	rm np_chk.o s1410.o

MacSCSI:
	cc -bu np_dvr.c
	cc -bu s1410.c
	ln -d -n MacSCSI -I 28 -R 40 np_dvr.o s1410.o -lc -o MacSCSI
	cp -f MSCSI TestSCSI
	cprsrc DRVR 28 MacSCSI TestSCSI
	rm np_dvr.o s1410.o

MountSCSI:
	cc mountscsi.c
	ln -mo MountSCSI mountscsi.o sys:lib/mixcroot.o -lc
	cprsrc DRVR 30 sys:system MountSCSI
	cp -f MountSCSI MSCSI
	cprsrc DRVR 28 MacSCSI MountSCSI
	rm mountscsi.o

BootSCSI:
	cc bootscsi.c
	ln -mo BootSCSI bootscsi.o sys:lib/mixcroot.o -lc
	cprsrc DRVR 30 sys:system BootSCSI
	rm bootscsi.o

/*
 * s1410.c version 1.0, July 20, 1985
 *
 * MacSCSI I/O routine for XEBEC S1410 with ST506 drive and similar
 * SASI/SCSI controllers that are pre-SCSI standard. Other controller
```

```
 * drive combinations will require changes. This version was developed
 * on the Mac under Aztec C.
 *
 * Copyright 1985 by John L. Bass, DMS Design
 * PO Box 1456, Cupertino, CA 95014
 * Right to use, copy, and modify this code is granted for
 * personal non-comercial use, provided that this copyright
 * disclosure remains on ALL copies. Any other use, reproduction,
 * or distribution requires the written consent of the author.
 *
 * Sources are available on diskette from Fastime, PO Box 12508
 * San Luis Obispo, Ca 93406 -- (805) 546-9141. Write for ordering
 * information on this and other Mac products.
 */

/*
 * SASI/SCSI Command and Sense Blocks
 */
struct scsicmd {
        char    sc_cmd;                 /* command code */
        char    sc_adrH;                /* High Byte of address */
        char    sc_adrM;                /* Middle byte of address */
        char    sc_adrL;                /* Low byte of address */
        char    sc_arg;                 /* count or interleave value */
        char    sc_vendor;              /* vendor byte -- seek algorithm */
};

struct scsisense {
        char    ss_code;                /* status code */
        char    ss_adrH;                /* address when valid */
        char    ss_adrM;
        char    ss_adrL;
};

/*
 * NCR5380 registers on the MacSCSI host adapter found at
 * location 0x500000
 */
struct NCR5380 {
        char wr_data,   filla;          /* force scsi bus data */
```

(Continued on page 101)

DDJ Classifieds

Software

$50 FORM & SCREEN PAINTER FOR dBASE, BASIC & TURBO
100,000 copies shipped with dBASE II, now on PC & MSDOS. "Paint" prompts, data fields, lines, boxes where you want them and ZIP® writes dBASE II/III, BASIC or Turbo Pascal code. Then add one line to your programs for professional data entry and reports. $50 cash, check, or VISA, no COD.
MAGNUM DATA INC.
627 South Plymouth Blvd.
Los Angeles, CA 90005
(213) 937-0808

Quality Software at Giveaway Prices!
We have to clear lots of CP/M & MSDOS software. Many 75% off list price. Source for many. Thousands of 5-inch & 8-inch diskettes—under $1 ea. "Everyman's Database Primer"/ "dBase II for Every Business"—$7.50 ea. + $2 S/H.
Software Salvage
Box 640
Norwalk, CT 06856

DBase II User—Converting to "C"
Try the dBx translation system— from DBase II to quality "C"; incl. translator, screen handler, & sort (w/source)— uses any file handler. For MSDOS, XENIX & UNIX Sys 5.
Desktop AI
Box 640
Norwalk, CT 06856

FED "Binary File Editor"
Allows the user direct access in both HEX and ASCII format to any disk file on the IBM PC. FED can patch object modules, examine word processing files, repair damaged files and verify the results of I/O operations. S/S DOS disk for 128 IBM PC. Only $49'' + $5 s/h''.
The Whitewater Group
2912 N. Burling Avenue
Chicago, IL 60657
(312) 975-6095

TECMAR GRAPHICS LIBRARY

TEKMAR lets you do high-res graphics on your TECMAR Graphics Master. Features windowing, viewporting, clipping, axis rotation. Similar to Tekronix graphics. Includes screen dump/restore. Epson screen print, support for HP and Western Graphtec plotters. Includes three curve-fitting programs and graphics application SOURCE CODE. Requires MS-FORT 3.20, or Lahey F77L.
Price: $195.
ADVANCED SYSTEMS CONSULTANTS,
18653 Ventura Boulevard, Suite 351,
Tarzana, California 91356
(818) 407-1059

ECLIPSE SYSTEMS

AZTEC C65 + ProDOS
Use Aztec C DOS 3.3 software under ProDOS. Run programs without relinking. System includes recursive Shell, support for pseudo-code, vi like editor with macros, library upgrade and source code. Requires ProDOS user's disk. $49.95.
Eclipse Systems, 223 Matthew Rd., Marion, PA 19066. (215) 667-8354

SOFTWARE SENTINEL
A hardware key that prohibits unauthorized use of software. Its benefits: Unlimited backup copies; unbreakable; site licensing control; no floppy required with hard disk; transparent; pocketsize. Evaluation Kit available. PC compatible.
Rainbow Technologies, Inc.
17971 Skypark Circle, Suite E
Irvine, CA 92714 (714) 261-0228

DBase II User—Converting to "C"?
Try the dBx translation system—from DBase II to quality "C"; incl. translator, screen handler, & sort (w/source)— uses any file handler. For MS-DOS, XENIX & UNIX sys 5.
Desktop AI
Box 640
Norwalk, CT 06856

Utility

PRODUCE ASSEMBLY CODE FAST!
Basic XPL is a high-level assembler with source code portability between CP/M and PCDOS. Complete with detailed 65-page manual. Satisfaction guaranteed. Send $79 check to author:
Gary D. Campbell
205 Sunbird Cliffs Lane
Colorado Springs, CO 80907

****Pascal's Friend****
PASCAL'S FRIEND v. II contains source code for use with IBM PC Turbo Pascal: 1-2-3 style menu routines, keyboard handling, save and restore screens, read disk directory or any track and sector, system clock and calendar routines, write strings in any attribute, DOS funcion calls and MORE!
J. S. Computing
815 N. 12th St.
Suite 5
Allentown, PA 18102
(215) 821-9020

Hardware

DIAL & ORDER
24 HR. MODEM LINE
(408) 425-4851
Specializing in technical/reference books and computer supplies.
Wise Dog Computerbooks
1310 Fair Ave, Santa Cruz, CA 95060

Changing your Address?
Send a note with your address label from a DDJ magazine cover and your new address to:
Dr. Dobb's Journal
2464 Embarcadero Way
Palo Alto, CA 94303

Dr. Dobb's Journal
is pleased to announce the
DDJ Classifieds

RATES: DISPLAY ADVERTISERS: Price per column inch $100. Ad must run in 3 consecutive issues.
LINE ADVERTISERS: Price per line $12 (35 characters per line including spaces). Minimum of 5 lines. 3 consecutive issues. Add $3 per line for boldface type. Add $25 if a background screen is desired.
DISCOUNTS: Frequency discounts for 6 consecutive ads (less 7%) and for 12 consecutive ads (less 15%).
MECHANICAL REQUIREMENTS: Camera ready art or typewritten copy only (phone orders excepted). Display: Specify desired size, include $20 if reduction is required. Line: Specify characters in boldface, caps. Allow 6 weeks for publication.
PAYMENTS: Full payment in advance. Check, money order, Visa, M/C, AmEx.

Run this ad in ______ issues
Size: ______ col x ______ inches or 1 col x ______ lines
Payment by: ______ check ______ m/o ______ Visa ______ M/C ______ AmEx
Card # ______________________ Exp Date ______
Signature ______________________
Print Name ______________________
Company ______________________
Address ______________________
City ______________ St ______________ Zip ______
Phone ______________ Current Subscriber ______

Mail to or phone Shawn Horst, DDJ Classifieds, 2464 Embarcadero Way, Palo Alto, CA 94303 (415) 424-0600

```
        char wr_icmd,   fillb;          /* initiator command */
        char wr_mode,   fillc;          /* ncr5380 mode */
        char wr_tcmd,   filld;          /* target command */
        char wr_sele,   fille;          /* select enable */
        char wr_send,   fillf;          /* start send operation */
        char wr_trec,   fillg;          /* start target recieve */
        char wr_irec,   fillh;          /* start initiator recieve */
        char fill00,    rd_data;        /* current scsi bus data */
        char fill01,    rd_icmd;        /* initiator command status */
        char fill02,    rd_mode;        /* ncr 5380 mode */
        char fill03,    rd_tcmd;        /* target command status */
        char fill04,    rd_bstat;       /* current bus status */
        char fill05,    rd_stat;        /* chip bus and status info */
        char fill06,    rd_input;       /* input data */
        char fill07,    rd_reset;       /* reset strobe */
};

/*
 * Phase defines        TCMD            BSTAT
 */
 #define        P_DOUT  0x00            /* 0x60 data out */
 #define        P_DIN   0x01            /* 0x64 data in */
 #define        P_CMD   0x02            /* 0x68 command */
 #define        P_STAT  0x03            /* 0x6C status */
 #define        P_MOUT  0x06            /* 0x70 Message Out */
 #define        P_MIN   0x07            /* 0x74 Message In */

/*
 * Global for drive size for use by format routine
 */
long    ScsiDrvSize = 4L * 17L * 153L;

/*
 * ScsiReset -- assume this is the only host adapter in system and do
 * a hard reset of the buss and controllers.
 */

ScsiReset() {
        long cntr;
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;

        ncr->wr_data = zero;
        ncr->wr_mode = zero;
        ncr->wr_tcmd = zero;
        ncr->wr_icmd = 0x80;
        for(cntr=0x40000;cntr>0;cntr--);
        ncr->wr_icmd = zero;
}

/*
 * ScsiCmd - Select the target controller 0, build and transfer
 * the command block.
 */
ScsiCmd(opcode,lun,blk,len,ctl) {
        struct scsicmd cmd;
        long cntr;
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;
        register char *ptr;
```

(Continued on next page)

Listing One

```
        ncr->wr_tcmd = zero;                                /* select controller */
        ncr->wr_data = 1;
        ncr->wr_icmd = 0x05;
        for(cntr=0x40000;cntr>0 && (ncr->rd_bstat & 0x40) == zero;cntr--);
        ncr->wr_icmd = zero;
        cmd.sc_cmd  = opcode;           /* build scsi command block */
        cmd.sc_adrH = lun<<5;
        cmd.sc_adrM = blk>>8;
        cmd.sc_adrL = blk;
        cmd.sc_arg  = len;
        cmd.sc_vendor = ctl | 1;        /* force XEBEC ST506 Halfstep */
        ScsiOut(6,&cmd,P_CMD);          /* send cmd to ctlr */
}

/*
 * ScsiOut/ScsiIn - transfer bytes on data bus with req/ack handshake
 * In the interest of speed we ignore edge following, the controller
 * will respond within a microsecond or so during a particular phase.
 * The rest of the loop is unfolded and optimized for the Aztec C
 * compiler to generate 9 memory references per byte. Best case would
 * be 7 memory references.
 */
ScsiOut(len,ptr,phase)
register char *ptr;
{
        register high = 0x11;
        register long low = 0x01;
        register char *i,*d;
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;

        d = &ncr->wr_data;
        i = &ncr->wr_icmd;
        ncr->wr_tcmd = phase;
        ncr->wr_icmd = 0x01;
        do {
                while((ncr->rd_bstat & 0x20) == zero);      /* sync with req */
                if((ncr->rd_stat & 0x08) == zero) break;    /* if done */
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
                while((ncr->rd_bstat & 0x20) == zero);       /* sync with req */
                if((ncr->rd_stat & 0x08) == zero) break;     /* if a cmdblk */
                *d = *ptr; ptr+=low; *i = high; *i = low;
                *d = *ptr; ptr+=low; *i = high; *i = low;
        } while ((len -= 8) > 0);
        ncr->wr_icmd = zero;
        return(0);
}

ScsiIn(len,ptr,phase)
register char *ptr;
{
        register high = 0x10;
        register long one = 0x01;
        register char *i,*d;
        register zero = 0;
```

(Continued on page 104)

Quelo® 68000 Software Development Tools

68000/68010 Assembler Package

Assembler, linker, object librarian and extensive indexed typeset manuals.

Conforms to Motorola structured assembler, publication M68KMASM[4]. Macros, cross reference and superb load map, 31 character symbols.

Optimized for CP/M-80, -86, -68K, **MS-DOS**, PC-DOS . $ 595

Portable Source in "C" until Nov. 1, 1985 . . . . . $1495
after Nov. 1, 1985 . . . . . $3000

Lattice® 68000 "C" Cross Compiler and Quelo 68000/68010 Assembler Package

Optimized for MS-DOS $1095

68200 Assembler Package

Optimized for CP/M-80, MS-DOS, PC-DOS . . . . . $ 595

68020 Assembler Package

First release Sept. 15, 1985 for **MS-DOS** . . . . . $ 750

For more information contact Patrick Adams

Quelo Inc.
2464 33rd W. Suite #173
Seattle, WA 98199
Phone (206) 285-2528
Telex II (TWX) 910-333-8171

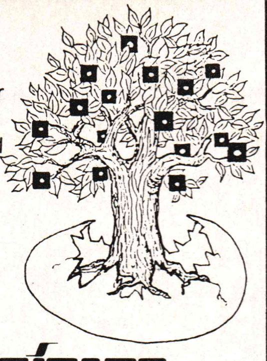
COD, Visa, MasterCard

CP/M, tm DRI.MS-DOS tm Microsoft. Lattice, tm Lattice Inc.

Circle **no. 72** on reader service card.

Tree Shell

A Graphic Visual Shell for Unix/Xenix End-Users and Experts Alike!



Dealer inquiries welcomed.

COGÍTATE

"A Higher Form of Software"
24000 Telegraph Road
Southfield, MI 48034
(313) 352-2345
TELEX: 386581 COGITATE USA

Circle **no. 24** on reader service card.

No source code for your REL files?

REL/MAC

converts a REL file in the Microsoft™ M80 format to an 8080 or ZILOG™ Z80 source code MAC file with insertion of all public and external symbols.

- **REL/MAC** makes MAC source files
- **REL/MOD** lists library modules
- **REL/VUE** displays the bit stream
- **REL/PAK** includes all of the above
- 8080 REL/MAC **demo disk** $10.00

REL/PAK for 8080 only $99.95
REL/PAK for Z80 & 8080 . . . . . $134.95
on 8"SSSD disk for CP/M™ 2.2

Send check, VISA, MC or C.O.D. to

MicroSmith
COMPUTER TECHNOLOGY
P.O. BOX 1473, ELKHART, IN 46515
1-800-622-4070
(Illinois only 1-800-942-7317)

Circle **no. 41** on reader service card.

Disk Sale

Dysan CORPORATION

TYPE	BOX OF 10
5"-SS/DD-48 TPI	19.50
5"-DS/DD-48 TPI	25.50
5"-SS/DD-96 TPI	29.50
5"-DS/DD-96 TPI	37.50
5"-DS/DD-IBM/AT	52.95
8"-SS/SD-48 TPI	23.95
8"-SS/DD-48 TPI	25.50
8"-DS/DD-48 TPI	29.95
3.5"-SS/DS	32.95

Available Soft or Hard Sector
For Plastic Case Add 1.25/Box
Plus Tax & Shipping
Cash, Visa, Mastercard, COD

Integral Systems Corp.
2900-H Longmire Drive
College Station, TX 77840
(409) 764-8017

Circle **no. 23** on reader service card.

TALK BACK to your PC ...and listen

TRUE SPEECH CAPABILITY FOR THE IBM PC! The DIALOG Voice System digitizes speech in real time, stores it on disk, and plays it back on command. Each board comes complete with the software drivers to hook our hardware to the applications program of your choice.

WHAT WILL IT DO FOR ME?
Applications include verbal annotation of text, electronic mail and messaging, intelligent phone management, telemarketing, and remote data entry. Dialogic provides the optimum speech I/O "engine" for the IBM PC world.

HOW DO I GET ONE?
DIALOG/1, the basic model, is priced at $295. DIALOG/2, with an intelligent phone interface and touch-tone decoding, is $495. DIALOG/3, with modem, is $595. Each board comes with a detailed user's manual, diskette, with the software drivers, and demo package.

Call or write us for detailed specifications:
DIALOGIC CORPORATION, 60 Baldwin Road
Parsippany, NJ 07054 (201) 334-8450
Demo Line (201) 334-1268

DIALOGIC

Talk back. With DIALOGIC.

Circle **no. 51** on reader service card.

NOW C HERE!
CROSS SOFTWARE for the NS32000

Also Available for IBM PC

INCLUDES:

★ Cross Assembler ★
★ Cross Linker ★
★ Debugger ★
★ N.S. ISE Support ★
★ Librarian ★
★ Pascal Cross Compiler ★
★ C Cross Compiler ★

U.S. prices start at $500

SOLUTIONWARE
1283 Mt. View-Alviso Rd.
Suite B
Sunnyvale, Calif. 94089
408/745-7818 ★ TLX 4994264

Circle **no. 118** on reader service card.

AT LAST —
A handbook that contains the Programming Codes for 100's of popular printers.

Announcing:

PROGRAMMERS' HANDBOOK OF COMPUTER PRINTER COMMANDS

The handbook gives you:

- Codes for printers made by over 40 Printer Manufacturers.
- Easy to use spiral bound book of over 250 pages of Programming Codes written in table form.
- Codes arranged by Written Code, Hex and Decimal equivalent, and with a brief description of what each code does.
- Codes for either Daisy-Wheel or Dot Matrix Printers (models through 1984).

ONLY $37.95 + $2 shpg./hdlg. ppd. with a two week approval guarantee. IF NOT SATISFIED, return in original carton for refund of book price only.

FOR MORE INFORMATION OR TO ORDER CALL OR WRITE:

Cardinal Point
INCORPORATED
(812) 876-7811 (9-5 EST)
P.O. BOX 596, ELLETTSVILLE, IN 47429
We accept MC, VISA, MO—same day shpg.
COD—$2 extra. CKs—Allow extra 14 days.

Circle **no. 11** on reader service card.

ICs PROMPT DELIVERY!!!
SAME DAY SHIPPING (USUALLY)

OUTSIDE OKLAHOMA: NO SALES TAX

640 Kbyte MOTHERBOARD KIT: IBM PC/XT Zenith 150 & 100, Compaq Portable & Plus: $73.10

QUANTITY ONE PRICES SHOWN

8087-3	MATH		$125.00
8087-2	COPROCESSORS		150.00
DYNAMIC RAM			
256K	256Kx1	120 ns	$ 3.95
256K	256Kx1	150 ns	2.95
64K	64Kx1	150 ns	1.10
EPROM			
27C256	32Kx8	250 ns	$17.50
27256	32Kx8	250 ns	8.50
27128	16Kx8	250 ns	3.30
27C64	8Kx8	200 ns	5.25
2764	8Kx8	250 ns	2.95
2732A	4Kx8	250 ns	2.75
STATIC RAM			
6264LP-15	8Kx8	150 ns	$4.75
6116LP-3	2Kx8	150 ns	1.95

OPEN 6½ DAYS: WE CAN SHIP VIA FED-EX ON SAT.

NO EXTRA COST ON F-EX SAT DELIVERY

MasterCard/VISA or UPS CASH COD
Factory New, Prime Parts μPoo
MICROPROCESSORS UNLIMITED
24,000 S. Peoria Ave., BEGGS, OK. 74421 **(918) 267-4961**

Prices shown above are for August 5, 1985
Please call for current prices. Prices subject to change. Please expect higher or lower prices on some parts due to supply & demand and our changing costs. Shipping & insurance extra. Cash discount prices shown. Orders received by 6 PM CST can usually be delivered to you by the next morning, via Federal Express Standard Air @ $6.00, or Priority One @ $15.00!

Circle **no. 64** on reader service card.

Dr. Dobb's Journal

Subscription Problems?

No Problem!

Give us a call and we'll straighten it out. Today.

Outside California
CALL TOLL FREE: 800-321-3333
Inside California
CALL: 619-485-6535 or 6536

Listing One

```
        register struct NCR5380 *ncr = 0x500000;

        d = &ncr->rd_data;
        i = &ncr->wr_icmd;
        ncr->wr_tcmd = phase;
        do {
                while((ncr->rd_bstat & 0x20) == zero);
                if((ncr->rd_stat & 0x08) == zero) break;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
                *ptr = *d; ptr+=one; *i = high; *i = zero;
        } while ((len -= 8) > 0);
        return(0);
}

/*
 * ScsiStat - get the last two bytes to finish a command sequence
 */
ScsiStat() {
        register zero = 0;
        register struct NCR5380 *ncr = 0x500000;
        register char *ptr;
        short stat;

        ptr = &stat;
        ncr->wr_tcmd = P_STAT;
        while((ncr->rd_bstat & 0x20) == 0);
        *ptr++ = ncr->rd_data;
        ncr->wr_icmd = 0x10;
        ncr->wr_icmd = zero;
        ncr->wr_tcmd = P_MIN;
        while((ncr->rd_bstat & 0x20) == 0);
        *ptr++ = ncr->rd_data;
        ncr->wr_icmd = 0x10;
        ncr->wr_icmd = zero;
        ncr->wr_tcmd = zero;
        return(stat);
}

/*
 * ScsiFmt -- Do a default drive format (ST506)
 */
ScsiFmt() {
        ScsiCmd(0x04,0,0,12,0);          /* issue format command */
        return(ScsiStat());                        /* return status */
}

/*
 * ScsiRead and ScsiWrite -- do the I/O for a specified sector
 * and the related data.
 */
ScsiRead(sector,data)
char *data;
{
```

```
        ScsiCmd(0x08,0,sector,1,0);      /* issue read command */
        ScsiIn(512,data,P_DIN);          /* transfer in data */
        return(ScsiStat());                   /* return status */
}
ScsiWrite(sector,data)
char *data;
{
        ScsiCmd(0x0A,0,sector,1,0);      /* issue write command */
        ScsiOut(512,data,P_DOUT);        /* transfer out data */
        return(ScsiStat());                   /* return status */
}

/*
 * MacSCSI non-partitioned format routine.
 * This version was developed on the Mac under Aztec C.
 *
 * Copyright 1985 by John L. Bass, DMS Design
 * PO Box 1456, Cupertino, CA 95014
 * Right to use, copy, and modify this code is granted for
 * personal non-comercial use, provided that this copyright
 * disclosure remains on ALL copies. Any other use, reproduction,
 * or distribution requires the written consent of the author.
 *
 * Sources are available on diskette from Fastime, PO Box 12508
 * San Luis Obispo, Ca 93406 -- (805) 546-9141. Write for ordering
 * information on this and other Mac products.
 */

struct Volume {
                short   drSigWord;       /* should be $D2D7 */
```

(Continued on next page)

Circle no. 86 on reader service card.

Circle no. 122 on reader service card.

```
                long    drCrDate;       /* date and time of initialization */
                long    drLsBkUp;       /* date and time of last backup */
                short   drAtrb;         /* volume attributes */
                short   drNumFls;       /* # of files in file directory */
                short   drDirSt;        /* first logical block of file dir */
                short   drBlLen;        /* # of logical blocks in file dir */
                short   drNmAlBlks;     /* # of allocation blocks on volume */
                long    drAlBlkSiz;     /* size of allocation blocks */
                long    drClpSiz;       /* # of bytes to allocate */
                short   drAlBlSt;       /* logical block number of first
                                                        allocation block */
                long    drNxtFNum;      /* next unused file number */
                short   drFreeBks;      /* # of unused allocation blocks */
                char    drVN;           /* length of volume name */
                char    drFill[512-37]; /* volume name & start of alloc. map */
};
struct  Volume  v;

char    zeros[512];                             /* block worth of nulls */

long    ScsiDrvSize;                    /* number of 512 byte sectors */

main() {
                char *p;
                int     i;

#ifdef hack
        printf("Scsi reset\n");
        ScsiReset();
        printf("Drive being formatted\n");
        if(ScsiFmt()) {
                printf("Format Failed\n");
                exit(1);
        }
#endif
        v.drSigWord = 0xd2d7;
        v.drCrDate  = v.drLsBkUp = 0;
        v.drAtrb    = 0;
        v.drNumFls  = 0;
        v.drNxtFNum = 1;
        v.drDirSt   = 4;
        printf("Directory Start:      %6.6d\n",v.drDirSt);

        v.drBlLen   = 28;
        printf("Directory BlkLen:     %6.6d\n",v.drBlLen);

        v.drAlBlSt = v.drDirSt + v.drBlLen;
        printf("First Allocation Blk: %6.6d\n",v.drAlBlSt);

        v.drAlBlkSiz = 512L * (ScsiDrvSize/640L + 1L);
        v.drClpSiz  = v.drAlBlkSiz;
        printf("Allocation BlockSize: %6.6ld\n",v.drAlBlkSiz);

        v.drNmAlBlks = (ScsiDrvSize-v.drAlBlSt)/(v.drAlBlkSiz>>9);
        v.drFreeBks = v.drNmAlBlks;
        printf("Allocation Blocks:    %6.6d\n",v.drNmAlBlks);

        for(v.drVN = 0,p="MacSCSI";*p;p++) v.drFill[v.drVN++] = *p;

        printf("Initializing Drive\n");
        ScsiWrite(0,zeros);
        ScsiWrite(1,zeros);
```

(Continued on page 109)

ConIX™

NOW ONLY $79.95!

If you think you're missing out on innovative software developments because nobody is writing for CP/M™-80, take a look at us. We've adapted UNIX™ features to CP/M like never before, and with the kind of professional, quality-controlled product that you deserve. That product is none other than the critically acclaimed ConIX Operating System.

ConIX can provide any 48K+ CP/M-80 or compatible system with I/O Redirection and Pipes (uses memory or disk), perfected User Areas, Command and Overlay Path Searching, Auto Screen Paging, 8Mb Print Buffering, 22 new SysCalls, Function Keys, "Virtual" disk system, Archiver (saves over 50% disk), extensive command language, 300+ variables, 100+ commands, pull-down menu, and much more! Uses as little as 1/2K RAM! Runs with CP/M for true data and software compatibility. Installs easily without any system mods!

The ConIX package lists at $165 and has been advertised and sold internationally to many enthusiastic customers since October 1983. As a special limited offer, we've lowered the price of the complete ConIX system by 50% to only $79.95! Don't miss this opportunity to bring your 8-bit micro back into the software revolution. Order your copy of ConIX today!

Price includes manual, 8" disk, and user support. 5¼" conversions available. Contact your local dealer, or buy direct and add shipping: $4.50 UPS, $10 Canada, $25 overseas. NY residents add sales tax.

Computer Helper Industries Inc.
P.O. Box 680 Parkchester Station, NY 10462
Tel. (212) 652-1786 (for information/orders)

"We're helping your computer work better for you!"

UNIX: AT&T Bell Labs, CP/M: Digital Research, ConIX: Computer Helper Ind.

Circle no. 22 on reader service card.

Circle no. 84 on reader service card.

C Tools for PC-DOS and MS-DOS

PROFESSIONAL C UTILITIES

If you are looking for a great implementation of **vi**, **make**, **lint**, or a professional screen generator at an affordable price, your search is ended. At $125 each the following utilities offer what we believe to be the best software value available. At 2 for $200, 3 for $300, and all 4 for $350 the savings are appreciable.

What if you don't agree after purchase that you've received good value? Then send what you don't like back and get a refund (see Return Policy).

SunScreen & Lint work with Aztec C, Computer Innovations C, and Lattice C. "Z" and UniTools work with any 8086 C Compiler.

"Z" vi editor $125

- Based on the Berkeley **vi** editor
- Terse, powerful commands
- Create new commands via macros
- Undo last command
- Sophisticated pattern matching
- Sophisticated search/replace
- Automatically indents C programs
- Remembers operands
- Ctags

UniTools make, diff, & grep $125

Create a control file that defines the operations and dependencies for creating a module. When you need to recreate the module simply execute **make**. Based on time and dates on files, **make** decides which modules to compile or assemble, which utilities to execute, and which parameters to use.

SunScreen $125

- Powerful utility for producing professional looking input/output screens.
- Simplifies maintenance
- Works with color and mono
- Field by field editing options
- Free format or column format
- Cursor left/right/up/down
- Split and multiple screens
- On screen error messages
- Produces C modules
- Build libraries of stock screens

Lint....................... $125

Lint analyzes programs for datatype errors, subroutine argument errors, portability problems.

RETURN POLICY

All software is returnable within 30 days for refund. Shipping costs are not refundable. A small restocking fee may be charged. Only items shipped within the USA can be returned.

DISCOUNTS

Order 2-3 items for a 20% discount (cost is $100 per item), 4-7 for a 25% discount ($93.75), 8 or more for a 30% discount ($87.50).

TO ORDER

Call 1-800-TECWARE (1-800-832-9273). Call 201-530-7997 from NJ or outside USA. Orders can be payed by COD, VISA, Master Card, or American Express. One and two-day delivery available at an additional charge. Delivery outside the USA is $10 + 2% of order.

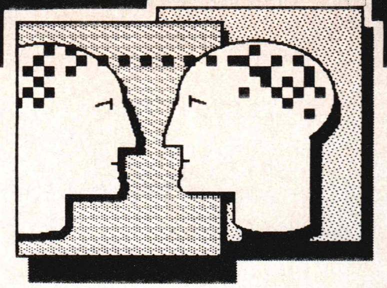
Orders may also be sent to: Manx Software Systems, P.O. Box 55, Shrewsbury, NJ 07701

Circle no. 87 on reader service card.

```
        if(ScsiWrite(2,&v)) {
                printf("Write of config block failed\n");
        }
        for(i=3;i<ScsiDrvSize;i++) if(ScsiWrite(i,zeros)) {
                printf("Write failed at block %d\n",i);
        }
        printf("Checking Drive\n");
        for(i=0;i<ScsiDrvSize;i++) if(ScsiRead(i,zeros)) {
                printf("Read failed at block %d\n",i);
        }
        printf("Format completed ok\n");
}
main()
{
        printf("Exit code %d\n",OpenDriver("\P.MacSCSI"));
}

main()
{
        *(int *)0x210 = 5;                /* boot drive to MacSCSI */
}
```

End Listing

Due to a lack of space in this issue the listing for the Mac SCSI disk driver (np_dvr.c) will be published in the October issue.



by Robert Blum

The CP/M *Exchange* RCP/M *system is available for your use 24 hours a day, 7 days a week. Reach it by dialing (404) 449-6588.*

Goodbye Old Friend

Effective July 1, 1985 Digital Research is changing the manner in which they support their products. Some products, the more fortunate ones, will continue to receive support in varying levels, while other, older and more mature products will no longer be supported.

Product support has been broken into four levels. The highest level of support a product can have is level "A", also called priority support. It was not explained exactly what this means, but I assume that any support calls made for a level "A" product will be given priority over other calls.

Level "B" support is active support. You may encounter some delays in receiving support for level "B" products. Priority is given to questions submitted on CompuServe; next phone calls will be answered; and finally letters will be answered.

Level "C" support is limited to questions submitted on CompuServe and phone calls. Level "D" support is for mature products that are not actively supported.

What does all this mean? To the CP/M Plus user it means that you won't be able to call DRI any longer expecting to get help, because CP/M Plus has been assigned support level "D". Effectively, this appears to eliminate any further hope of buying an unconfigured copy of CP/M Plus. I doubt that dealers will continue to sell a product that isn't supported by the manufacturer. A few hardware manufacturers are still packaging it with their machines; hopefully they will also be able to support it.

Fortunately for the one million or so of us using CP/M 2.2, it was assigned to level "C" support. Some time still remains for the once most popular operating system for microcomputers. When support is finally dropped on CP/M 2.2 I doubt that any great impact will be felt. There are hundreds of reference books available for it and a good supply of heavily experienced programmers knowledgeable on the system.

AMPRO Little Board

In the May 1985 issue of *DDJ* Richard Conn reviewed the AMPRO Little Board and Bookshelf Computers. Since the electronic version of this column is running 24 hours a day on an AMPRO Little Board, I have decided to offer some impressions of this computer that I have formed as a user.

Measuring a mere 5¾-inches × 7¾-inches, the AMPRO Little Board is a complete 64K Z-80 computer on a single board. Its size, in fact, is the same as a mini-floppy disk drive; the mounting tabs on the board are spaced to match the mounting holes on any standard drive. The input/output facilities include two asynchronous serial RS-232 ports, a Centronics compatible parallel interface, and a single chip floppy disk controller capable of handling up to four 5¼-inch drives. To round out the package is a well-implemented CP/M 2.2 enhanced by ZCPR3 and a large set of utility programs.

All that remains to form a complete system is to supply a meager 5 watts of power, a CRT terminal attached to one of the serial ports, and mini-floppy disk drives of your choice. Finally, a cabinet would probably be desirable to provide protection for the delicate components.

If your need for external storage exceeds that provided by the floppy disk drives, a SASI/SCSI peer buss daughter board is available as an option. This board, along with an updated BIOS and several utility programs, will allow a hard disk to be attached to the Little Board. When used, this option can account for up to 20 Mb of additional storage.

I don't pretend to be an electrical engineer and I don't have any inside information from the people who designed the Little Board. It's apparent to me, however, that the principles of reliability and cost effectiveness were held above all others when this computer was designed.

The component density, ICs per square inch of board space, is far less than what I am accustomed to. Even though this is not documented, I believe that forced air cooling of the Little Board is not necessary if adequate air circulation and normal room temperature is maintained. Better yet, if a cabinet with forced air cooling is used, component life should not be adversely affected by temperature extremes. And, even when the board is attached directly to the bottom of a floppy disk drive, heat should not be a problem.

Serial Interface

The Z80 Dual Asynchronous Receiver/Transmitter (DART) is used to implement the two serial ports. This device is the little brother to the Z80 SIO. They are practically identical in all ways including software programming. So similar, in fact, that they can be interchanged in the same socket. The difference that exists between the two is the DART's inability to handle synchronous transmission protocols such as BI-SYNC and SDLC.

Operation and setup of the DART is handled completely through software.

Programming is accomplished by loading the desired transmission parameters into one or more of the five write registers contained on the chip. For instance, the common transmission characteristics of word length, number of stop bits, and baud rate are all completely programmable through software by loading the proper values into write registers four and five. The more advanced features, enabling interrupts and setting interrupt vectors, are handled in like manner.

Both serial channels can be programmed for a full range of baud rates. Channel A can be used up to a maximum of 38.4K baud, which to my knowledge is fast enough to drive any commercially available CRT at its maximum rate. Channel B is limited to 9600 baud, however. Nevertheless, I don't think the lower baud rate will present any practical limitations since most auxiliary devices, modems and printers, are unable to receive data faster than at 9600 baud.

The otherwise complete serial section is limited, however, by the number of signal lines that are brought out to the mating connector. In addition to the input and output data lines and two ground lines, only two control lines, one input and one output, are made available. In any other than the most simple application, a single flow control line may prove to be inadequate.

Parallel Interface

The Centronics compatible parallel interface is implemented with a single 8 input D-type latch. Here again, the number of control lines is limited. Only the peripheral device busy line is brought in for flow control. The other control lines, paper empty etc., are conspicuously missing.

Floppy Disk Controller

The floppy disk controller used on the Little Board is the Western Digital 1770. It is similar to its predecessor, the WD179X, but also contains a digital data separator and write precompensation circuity. The most noticeable change, however, is in its size; the WD1770 is completely contained in a single 28 pin package. Only a few years ago several inches of

tightly packed board space were needed to implement a floppy disk control circuit. The 1770 represents a dramatic reduction in chip count and overall board complexity that should result in improved reliability.

If there is a drawback to the Little Board disk controller it is its inability to control 8-inch drives. At first I thought this would be a big problem. Then I experienced the over 800K capacity of a 96TPI format 5¼-inch disk.

Booting Up

Starting up the Little Board for the first time is a snap. After attaching the floppy disk drives with a standard 34 pin ribbon cable, all you need to do is to attach a CRT terminal set to 9600 baud to serial channel A. There are no other special requirements. Simply insert the CP/M disk into disk drive A and hit reset. The standard monitor ROM will load the operating system from disk and begin its operation. Nothing could be easier.

Once the system is up and running, you may want to change the default serial port parameters or activate an option that automatically loads a program at cold start. A reconfiguration program is provided to enable you to make these changes without going through the hassle of editing and reassembling the BIOS portion of CP/M.

Another option of the Little Board that I find most useful is the ability to reconfigure dynamically one of the four floppys to that of another computer. For example, often I need to read disks formatted on a Morrow system. To do this I simply assign drive E: to one of my physical drives and set the disk parameters to simulate the Morrow system. Any further access to drive E: now causes CP/M to respond as if it were actually running on the other computer.

Two associated utility programs are provided to use the drive E: option. One will allow you to set automatically drive E: to any of about a dozen different disk types simply by selecting it from the menu. The other prompts you for the various disk parameters used by CP/M.

A host of other utility programs are provided with the Little Board. I have found all of them to be well implemented and very useful.

User Impression

I have been using the Little Board daily for several months. And over the last three weeks it has been the main processor for my 24 hour per day Bulletin Board System. During that time the Little Board has not been down a single moment.

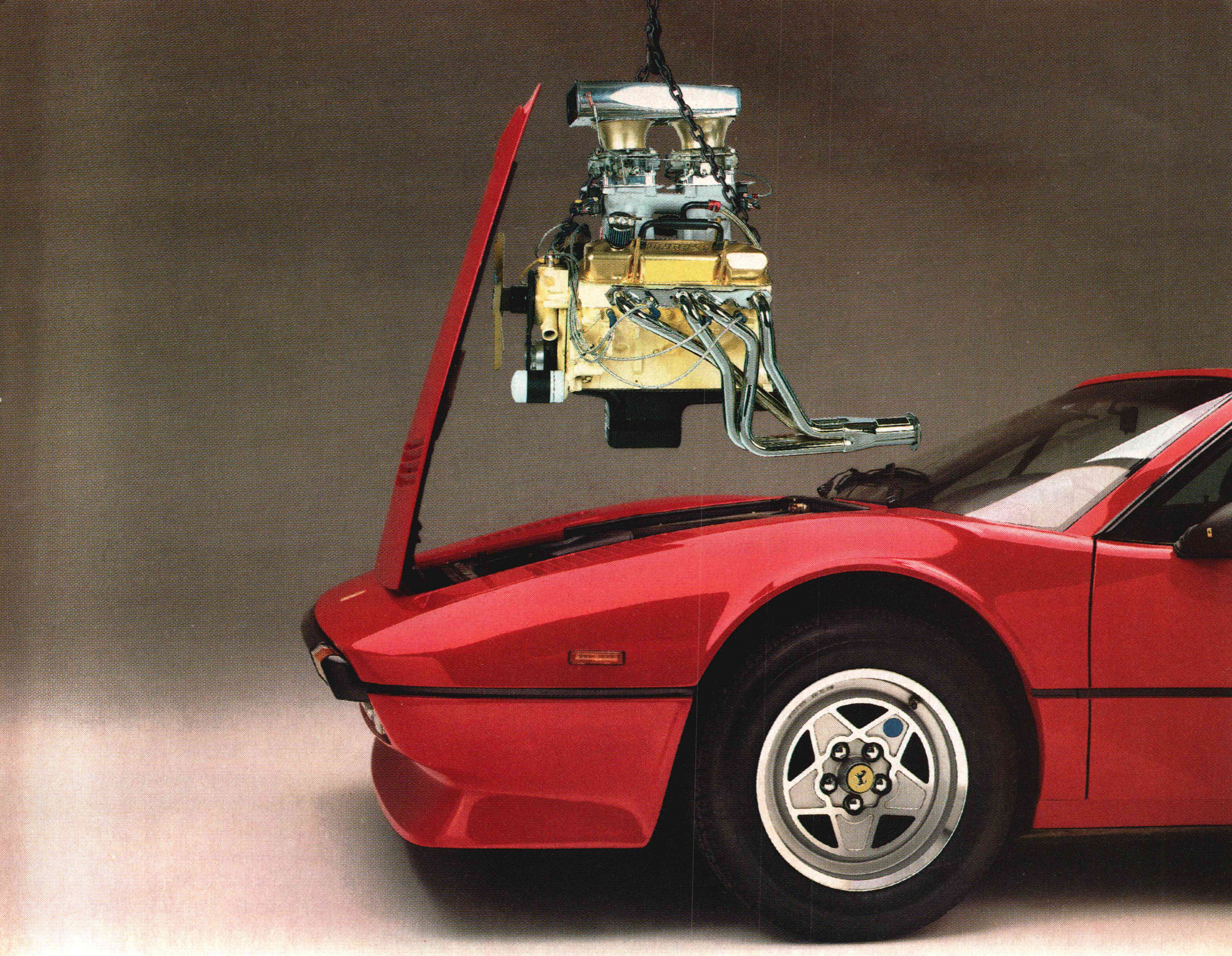
The only fault that I find significant enough to mention is the failure to bring all the serial and parallel control lines out to their connectors, although, in all fairness, I haven't needed them. As I mentioned earlier, I have attached a modem and a serial printer to channel B without problem.

Wrapping Up

I haven't been able to finish everything that I wanted to say about the Little Board; I'm working on adding a real time clock and disk data interrupts to the BIOS. That material will have to wait until next month.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 199.**



AT™ Pfantasies for your PC or XT.™

Want better speed and memory on your PC or XT without buying an AT?

You've got it!

Phoenix's new Pfaster™286 co-processor board turns your PC or XT into a high-speed engine 60 percent faster than an AT. Three times faster than an XT. It even supports PCs with third-party hard disks. But that's only the beginning.

You can handle spreadsheets and programs you never thought possible. Set up RAM disks in both 8088 and 80286 memory for linkage editor overlays or super-high-speed disk caching. All with Pfaster286's 1mb of standard RAM, expandable to 2mb, and dual-mode design.

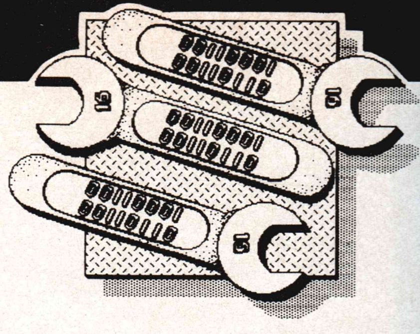
You can develop 8086/186/286 software on your XT faster. Execute 95 percent of the application packages that run on the AT, excluding those that require fancy I/O capabilities your PC or XT hardware just isn't designed to handle. Queue multi-copy, multi-format print jobs for spooling. Or, switch to native 8088 mode to handle hardware-dependent programs and back again without rebooting. All with Pfaster286's compatible ROM software. And, Pfaster286 does the job unintrusively! No motherboard to exchange. No wires to solder. No chips to pull. Just plug it into a standard card slot, and type the magic word, "PFAST."

If you really didn't want an AT in the first place, just what it could do for you, call or write: Phoenix Computer Products Corp., 1416 Providence Highway, Suite 115, Norwood, MA 02062; (800) 344-7200. In Massachusetts, 617-762-5030.

Programmers' Pfantasies™ by Phoenix

XT and AT are trademarks of International Business Machines Corporation. Pfaster286 and Programmers' Pfantasies are trademarks of Phoenix Computer Products Corporation.
For the Ferrari aficionado: yes, we know this is a rear engine car. We are showing the addition of a second engine to symbolize how Pfaster can be added to your PC or XT to increase performance.

Circle **no. 91** on reader service card.



by Ray Duncan

The programs published in this month's column are available for downloading from the Laboratory Microsystems RBBS at (213)306-3530 (300 or 1200 baud).

Laserjet Redux

After the June 1985 16-Bit Software Toolbox appeared in print, I received some "hot and bothered" phone calls from friends at Hewlett-Packard about the tone of my comments on the Laserjet Printer. They knew very well that I consider the Laserjet to be a fantastic printer and the greatest thing since sliced bread, but they thought that readers might get an altogether different impression by reading the column. Upon reviewing the June column, I'm afraid I must agree with them.

First, the recommendation to switch to Spellbinder was strictly that of the dealer and is not an official position of Hewlett-Packard. Furthermore, Hewlett-Packard distributes a technical bulletin to its dealers with information on converting IBM PC WordStar for use with the Laserjet Printer (though the conversion doesn't offer as many capabilities as the specially patched version I got from MicroPro Technical Support; but see page 4), and the HP-150 and HP-110 versions of WordStar support the Laserjet directly. Finally, the slow speed (2–3 pages/minute) is the result of the strategy used by IBM PC WordStar for microjustification; the Laserjet is capable of 8 pages/minute when driven by other word processors.

As a gesture of atonement to Laserjet fans, this month's Listing One (page 117) is a file-printing program in Lattice C that prints two pages across in "Landscape" mode along with filenames, a time and date stamp, and page numbers.

We should note in passing that the new version of Microsoft Word offers full support for the Laserjet and for the IBM Enhanced Graphics Adaptor; between the three, it should be possible to turn out some excellent-looking documents.

Microsoft Assembler

In the June 1985 column, I also inadvertently introduced some confusion about the IBM Macro Assembler Version 2. Although *my* IBM Product Center didn't know about it, there actually was a reduced price update offer from IBM for owners of the Macro Assembler Version 1.0; it involved sending $75 and the cover sheet from your original manual to IBM, and in return they would ship you the new assembler and documentation. This update offer expired at the end of June 1985.

I have since learned that the reference section on the assembler mnemonics, which I complained had disappeared from the manual, was expanded into another volume called the *Macro Assembler Programmer's Reference*. This second manual is supposed to be delivered to you along with the operations manual and the disk, but none of the stores I've been to are displaying it with the Macro Assembler—so if you want it, you'll have to know to ask for it.

The respective version numbers used by IBM and Microsoft are also causing a lot of confusion. The IBM Macro Assembler Version 2.0 appears to be essentially the same as the Microsoft Macro Assembler Version 1.25, released over a year ago. The SALUT utility is apparently IBM's sole addition to the package. There never was a Microsoft Macro Assembler Version 2.0; Microsoft jumped directly to Macro Assembler Version 3 with its latest release in order to make the version number match the current release of MSDOS/PCDOS.

David Rabbers writes: "Microsoft MASM Version 3 is greatly worth the purchase. It not only includes a better assembler (implements the rest of the 80286 op-codes), which they claim has fewer bugs (at least some old ones are fixed and I haven't found the new ones yet), but you get two new utilities that alone are worth the $75 upgrade fee. First is a version of the Unix utility MAKE. I have never seen MAKE before, but this one seems simplistic though adequate for anything I can envision. The second utility is SYMDEB, a very good symbolic debugger. It is an enhancement to DEBUG with a natural and compatible (well, pretty close) extension to the DEBUG command syntax . . . . " SYMDEB can handle symbol tables generated by the Microsoft high-level language compilers.

Readers have reported perplexing problems when trying to convert to the IBM Macro Assembler 2.0. A number of programs that assembled and ran properly with IBM Macro Assembler 1.0 won't assemble without errors—or if they do assemble without errors, they won't run. An example of this is the statement:

```
name1 EQU (THIS BYTE)-
    (OFFSET name2)
```

This does not give an error message in itself (it assembles, strangely enough, as an equals sign followed by absolutely nothing), but it causes a cascade of phase errors throughout the rest of the program. Everything works fine when the statement is changed to:

```
name1 EQU $-name2
```

Steve Itzkowitz reported another odd experience working with the program DSKWATCH, which is available on many IBM PC bulletin boards. It appears to assemble identically with both IBM Macro Assemblers 2.0 and 1.0. However, with Microsoft Macro Assembler 1.27 and 3.0, it will not assemble cleanly, and when the errors are corrected by a trial and error process, the resulting program does not run. Other readers are invited to send in their own experiences and comments.

Detecting Intel Numeric Coprocessors

When writing 8086 applications that perform floating-point arithmetic, you should test for the presence of an Intel 8087 and exploit it if it is available. The accepted strategy for this test has been to initialize the processor, and then to write the control word into a previously zeroed memory location and examine the result:

```
fninit
mov    variable,0
fnstcw variable
. . .
cmp    variable,03ffh
jeq    np_present
jmp    np_absent
```

If an 8087 is present, the control word will be stored into the memory variable as 03FFH; if it is not present, the contents of the variable will not change. It is important not to use any wait instructions during this test: if the 8087 is not present, the wait will cause the 8086 simply to hang (this is why the **fninit** and **fnstcw** variations of the Intel mnemonics are used above, instead of **finit** and **fstcw**).

It turns out that programs using this strategy will fail to detect the presence of an 80287 in a PC/AT or other 80286-based system. Although the 80287 is software compatible with the 8087 in every other way, it initializes one of the bits in the lower byte of its control word opposite to the 8087. So the compare instruction in the above code sequence should be changed to

```
cmp byte ptr variable+1,3
```

Outstanding Software Value

I have been using a program called UNIFORM by Micro Solutions on my Kaypro IV for two years to read and write disks in the various incompatible CP/M 5¼-inch disk formats and have been very happy with it. It has saved me untold dollars in downloading fees. Now Micro Solutions has outdone itself with a release of UNIFORM for the IBM PC that ought to serve as a standard of quality, documentation, and ease of use for other software vendors.

Basically, UNIFORM for the PC consists of an installable system driver (UNIFORM.SYS) and a separate control and disk initialization program (UNIFORM.EXE). When the driver is linked into your operating system by including its name in the CONFIG.SYS file on the boot disk, it causes your second floppy disk drive (B:) to double as a logical drive (C:). When the drive is addressed as B:, it appears to be a normal MSDOS disk drive. But when it is addressed as C:, it almost magically makes disks in CP/M formats appear as though they were MSDOS disks!

You can do directories on CP/M disks, copy files to and from your other MSDOS format drives, erase or rename files, and in general completely ignore the fact that the disk in physical drive B: is formatted and structured in a way that is totally alien to the MSDOS operating system. In my opinion, this is a real software coup.

The separate control program is menu driven and allows you to set up the logical drive C: for some 80 different soft-sectored 5¼-inch CP/M formats, including the Epson QX-10, the Kaypro, and the DEC VT-180. In addition, if your system has a special controller for 80-track 5¼-inch drives or 8-inch drives, UNIFORM can handle that too, which gives you an even wider selection of disk formats to choose from. UNIFORM can also initialize disks in all of these different formats. This is a boon to small software houses—it allows them to centralize their distribution files on a large fixed disk hooked to an MSDOS machine, without dropping support for their CP/M customers. UNIFORM for the IBM PC costs $69.95 (it's easily worth ten times that) and is available from Micro Solutions at 125 South Fourth Street, DeKalb, Illinois 60115. Phone (815)756-3411.

Breaking with MSDOS

The MSDOS Control-Break handler, whose address is stored in the INT 23H vector, is a continual thorn in the side of application programmers. Whenever MSDOS detects a Control-C in a character stream (especially an unintentional or deliberate Control-C entered at the keyboard by the operator) during an I/O operation, it transfers directly to this handler. The default action of this handler is simply to blow the unsuspecting application out of the water . . . leaving temporary files on the disk, critical data files unclosed, critical interrupt vectors unrestored, etc. This, of course, is antithetical to the concept of a robust application, which should *always* be able to accomplish a graceful exit.

The programmer can take two approaches to the Control-C problem. The first is to disable and/or avoid Control-C detection by a variety of

methods, such as using only those I/O function calls that are not Control-Break sensitive; disabling Control-C detection during miscellaneous MSDOS operations with the MSDOS function call 33H; and putting all of the character drivers into "raw I/O" or "uncooked" mode.

A more straightforward way to beat the Control-C rap is simply to replace the MSDOS INT 23H handler with your own. On the IBM PC, you can also take over the INT 1BH vector, which is called by the ROM BIOS keyboard driver whenever a Control-Break is detected. This is easy enough for applications written in assembly language but a little more tricky for those using high-level languages. Note that the INT 23H handler is a standard MSDOS feature, and code that captures this interrupt is portable across all MSDOS systems. INT 1BH, however, is a function of the IBM ROM BIOS and will not be valid on other machines unless they are a very close IBM compatible.

Listing Two (page 119) is the source code for assembly language functions that can be linked with Lattice C applications to take control of the Control-Break interrupt handlers. This code should be readily portable to other C compilers and to assembly language applications. The function "capture" is called with the address of an integer variable within the C program; it saves the address of the variable, points the INT 1BH and 23H vectors to its own interrupt handler, and returns. When a Control-C or Control-Break is detected, the interrupt handler sets the integer variable within the C program to "True" and returns—the program can then poll this variable at its leisure. The function "release" simply restores the INT 1BH and INT 23H vectors to their original values, thereby disabling the application's interrupt handler. Listing Three (page 120) is a short C program that illustrates the use of these functions.

It is a good idea to use the MSDOS Get Interrupt and Set Interrupt function calls to inspect and modify the contents of the interrupt vectors. This will keep your application compatible with multitasking environments such

16-Bit (Text begins on page 114)

Listing One

```
 /**
  **
  **     LJ.C -- A printing utility for the HP LaserJet
  **
  **     This program prints a series of files on the LaserJet
  **     printer.   The files are printed in a "landscape" font at
  **     17 characters to the inch.    To take advantage of this
  **     density, two "pages" of information from the file are
  **     printed on each piece of paper (left and right halves).
  **
  **     Usage is:        LJ  file1 file2 file3 ...
  **
  **     Where file# is a valid MS-DOS filename, included on the
  **     command line.  This program is compatible with Lattice C
  **     on the IBM PC and the HP Touchscreen computers.
  **
  **     Joe Barnhart    original version            May 5, 1985
  **     Ray Duncan      date and time stamping      May 22, 1985
  **     Joe Barnhart    revised date stamping       June 6, 1985
  **
  **/

 #include <h\stdio.h>

 #define MAXLINE 56                   /* maximum lines per page */
 #define PAGE    '\f'                 /* for compilers without '\f' */
 #define TAB     8                    /* width of one tab stop */

 typedef struct {
             int ax, bx, cx, dx, si, di;
         } REGSET;

 main(argc, argv)
     int   argc;
     char  *argv[];
 {
     int   filenum;
     FILE  *fp, *prn, *fopen();

     if( ( prn = fopen( "PRN:", "w" ) ) == NULL )
         printf( "Error opening printer as file.\n" );
     else {
         /* initialize the LaserJet for landscape printing */
         fprintf( prn, "\033E\033&l1O\033(s17H\033&l8d6E" );
         for( filenum = 1; filenum < argc; filenum++ ) {
             fp = fopen( argv[filenum] ,"r" );
             if( fp == NULL )
                 printf( "File %s doesn't exist.\n", argv[filenum] );
             else {
                 printf( "Now printing %s\n", argv[filenum] );
                 printfile( fp, prn, argv[filenum] );
                 fclose( fp );
             }
         }

        fprintf( prn, "\015\033E" );              /* clear LaserJet */
     }
 }

printfile(fp,prn,filename)
    FILE  *fp,*prn;
    char  *filename;
{
    int pagenum = 1;

    while( !feof( fp ) ) {
        fprintf( prn, "\033&a0r85m5L\015" );    /* set left half */
        printpage( fp, prn );                   /* print page */
        if( !feof( fp ) ) {                     /* if more .. */
            fprintf( prn, "\033&a0r171m91L" );  /* set right half */
            printpage( fp, prn );               /* print another */
        }
        stamp( prn, filename, pagenum++ );      /* title */
        fputc( PAGE, prn );                     /* kick paper */
    }
}

printpage(fp,prn)
    FILE  *fp,*prn;
{
    char  c;
    int   line,col;
```

(Continued on next page)

as TopView and MS Windows. The interrupt vector table belongs to the operating system, not to the application, and it's not nice to modify memory directly that doesn't belong to you. If the fabled "Big DOS" ever comes along that runs in 80286-protected mode, your application will just get aborted if you try to twiddle the vector table without using the proper operating system services provided for that purpose.

Although in this little example program the INT 23H vector is restored by the "release" function, MSDOS will restore this vector for you automatically when your program exits. However, because INT 1BH is an IBM PC-specific interrupt handler and so is not known to MSDOS, it is *absolutely mandatory*, should your program modify this vector, that it restore the vector properly before exiting. Otherwise, the vector will be left pointing to some random area in the next program that runs—and the next time you press Control-Break, a system crash is the best you can hope for.

DDJ

16-Bit (Listing Continued, text begins on page 114)

Listing One

```
    line = col = 0;
    while( line < MAXLINE )
        switch( c = fgetc(fp) ) {
            case '\n':                          /* newline found */
                col = 0;                        /* zero column */
                line++;                         /* adv line cnt */
                fputc('\n',prn);
                break;
            case '\t':                          /* TAB found */
                do
                    fputc('\040',prn);
                while ( (++col % TAB) != 0 );
                break;
            case PAGE:                          /* Page break or */
            case '\377':                        /* EOF found */
                line = MAXLINE;                 /* force terminate */
                break;
            default:                            /* no special case */
                fputc(c,prn);                   /* print character */
                col++;
                break;
        }
}

stamp( prn, filename, pagenum )
    FILE  *prn;
    char  *filename;

    int   pagenum;
{
    char  datestr[10], timestr[10];

    fprintf( prn, "\033&a51171M" );             /* widen margins */
    fprintf( prn, "\015\033&a58R" );            /* move to row 58 */
    fprintf( prn, "File: %-133s", filename );
    fprintf( prn, "Page %-3d", pagenum);
    timestamp( timestr );
    datestamp( datestr );
    fprintf( prn, "   %s   %s", datestr, timestr);
}


datestamp( datestr )
    char  *datestr;
{
    REGSET regs;
    int  month, day, year;

    regs.ax = 0x2a00;
    int86( 0x21, &regs, &regs );
    month = ( regs.dx >> 8 ) & 255;
    day   = regs.dx & 255;
    year  = regs.cx - 1900;
    sprintf( datestr, "%02d/%02d/%02d", month, day, year);
}

timestamp( timestr )
char    *timestr;
{
    REGSET  regs;
    int  hours, mins;

    regs.ax = 0x2c00;
    int86( 0x21, &regs, &regs );
```

```
    hours = ( regs.cx >> 8 ) & 255;
    mins  = regs.cx & 255;
    sprintf( timestr, "%02d:%02d", hours, mins );
}
```

End Listing One

Listing Two

```
BREAK.ASM Control-C Handler for Lattice C

        title   Control-Break handler for Lattice C programs
        name    break
        include dos.mac

;
; Control-Break Interrupt Handler for Lattice C programs
; running on IBM PCs (and ROM BIOS compatibles)
;
; Ray Duncan, May 1985
;
; This module allows C programs running on the IBM PC
; to retain control when the user enters a Control-Break
; or Control-C.  This is accomplished by taking over the
; Int 23H (MS-DOS Control-Break) and Int 1BH (IBM PC
; ROM BIOS Keyboard Driver Control-Break) interrupt
; vectors.  The interrupt handler sets an internal
; flag (which must be declared STATIC INT) to TRUE within
; the C program; the C program can poll or ignore this
; flag as it wishes.
;
; The module follows the Lattice C parameter passing
; conventions, and also relies on the Lattice file DOS.MAC
; for the definition of certain constants and macros.
;
; The Int 23H Control-Break handler is a function of MS-DOS
; and is present on all MS-DOS machines, however, the Int 1BH
; handler is a function of the IBM PC ROM BIOS and will not
; necessarily be present on other machines.
;
        if      lprog
args    equ     6                   ;offset of arguments, Large models
        else
args    equ     4                   ;offset of arguments, Small models
        endif

cr      equ     0dh                 ;ASCII carriage return
lf      equ     0ah                 ;ASCII line feed

        pseg

        public  capture,release ;function names for C

;
; The function CAPTURE is called by the C program to
; take over the MS-DOS and keyboard driver Control-
; Break interrupts (1BH and 23H).  It is passed the
; address of a flag within the C program which is set
; to TRUE whenever a Control-Break or Control-C
; is detected.  The function is used in the form:
;
;               static int flag;
;               capture(&flag)
;

capture proc    near                ;take over Control-Break
        push    bp                  ;interrupt vectors
        mov     bp,sp
        push    ds
        mov     ax,word ptr [bp+args]
        mov     cs:flag,ax          ;save address of integer
        mov     cs:flag+2,ds        ;flag variable in C program
                                    ;pick up original vector contents
        mov     ax,3523h            ;for interrupt 23H (MS-DOS
        int     21h                 ;Control-Break handler)
        mov     cs:int23,bx
        mov     cs:int23+2,es
        mov     ax,351bh            ;and interrupt 1BH
        int     21h                 ;(IBM PC ROM BIOS keyboard driver
        mov     cs:int1b,bx         ;Control-Break interrupt handler)
        mov     cs:int1b+2,es
        push    cs                  ;set address of new handler
        pop     ds
        mov     dx,offset ctrlbrk
        mov     ax,02523H           ;for interrupt 23H
        int     21h
```

(Continued on next page)

Listing Two

```
            mov     ax,0251bH           ;and interrupt 1BH
            int     21h
            pop     ds                  ;restore registers and
            pop     bp                  ;return to C program
            ret
capture endp


;
; The function RELEASE is called by the C program to
; return the MS-DOS and keyboard driver Control-Break
; interrupt vectors to their original state.  Int 23h is
; also automatically restored by MS-DOS upon the termination
; of a process, however, calling RELEASE allows the C
; program to restore the default action of a Control-C
; without terminating.  The function is used in the form:
;
;                   release()
;
release proc        near                ;restore Control-Break interrupt
                                        ;vectors to their original state
            push    bp
            mov     bp,sp
            push    ds
            mov     dx,cs:int1b         ;set interrupt 1BH
            mov     ds,cs:int1b+2       ;(MS-DOS Control-Break
            mov     ax,251bh            ;interrupt handler)
            int     21h
            mov     dx,cs:int23         ;set interrupt 23H
            mov     ds,cs:int23+2       ;(IBM PC ROM BIOS keyboard driver
            mov     ax,2523h            ;Control-Break interrupt handler)
            int     21h
            pop     ds                  ;restore registers and
            pop     bp                  ;return to C program
            ret
release endp


;
; This is the actual interrupt handler which is called by
; the ROM BIOS keyboard driver or by MS-DOS when a Control-C
; or Control-Break is detected.  Since the interrupt handler
; may be called asynchronously by the keyboard driver, it
; is severely restricted in what it may do without crashing
; the system (e.g. no calls on DOS allowed).  In this
; version, it simply sets a flag within the C program to
; TRUE to indicate that a Control-C or Control-Break has
; been detected; the address of this flag was passed
; by the C program during the call to the CAPTURE function.
;
ctrlbrk proc        far                 ;Control-Break interrupt handler
            push    bx                  ;save affected registers
            push    ds
            mov     bx,cs:flag          ;set flag within C program
            mov     ds,cs:flag+2        ;to "True"
            mov     word ptr ds:[bx],-1
            pop     ds                  ;restore registers and exit
            pop     bx
            iret
ctrlbrk endp

flag        dw      0,0                 ;long address of C program's
                                        ;Control-Break detected flag
int23       dw      0,0                 ;original contents of MS-DOS
                                        ;Control-Break Interrupt 23H
                                        ;vector
int1b       dw      0,0                 ;original contents of ROM BIOS
                                        ;keyboard driver Control-Break
                                        ;Interrupt 1BH vector

            endps

            end
```

TRYBREAK.C Demo of Control-C Handler

```
/*

            Demonstrate Control-Break interrupt handler for Lattice C
            Ray Duncan, May 1985

*/
```

```
#include <h\stdio.h>

main(argc, argv)
    int   argc;
    char  *argv[];

{   int hit = 0;
    int c = 0;
    static int flag = 0;

    capture(&flag);
    puts("\nTRY has CAPTURED interrupt vectors\n");

    while ( (c&127) != 'Q')
    {   hit = kbhit();
        if (flag != 0)
                { puts("\nControl-Break detected\n");
                  flag=0;
                }
        if (hit != 0)
         {  c=getch();
            putch(c);
         }
    }
   release();
   puts("\n\nTRY has RELEASEd interrupt vectors\n");

}
```

End Listings

Circle **no. 69** on reader service card.

OF INTEREST

by Alex Ragen

IBM PC

Computers like the Compaq used to be called "portables," but that misnomer soon gave way to the more accurate "transportable" and more recently to "lugable," which is really the only way to describe those 30 pound monsters. There are now available, however, several machines that are truly portable.

Morrow introduced its Pivot II at Comdex in Atlanta. The original Pivot had a 16-line display, but the new model has a full 25-line display. There are three versions, priced between $1995 and $3795 depending on various optional features.

The big problem with portables of this type is the screen. Data General has changed screens twice on its impressive (but expensive) DG/1 in response to complaints about readability. The Pivot II uses a technology Morrow calls LumiCon, which combines the high quality of electroluminescent displays with the low cost and power requirements of liquid crystal displays (LCDs). Also a proprietary nonglare material makes the screen less susceptible to glare than the standard CRT display, Morrow claims.

The CPU is the 80C88, a CMOS version of the 8088, and the minimum memory configuration is 256K. The computer's dimensions are 13 × 6 × 9.5 inches, and it weighs only 13 pounds with two 5¼-inch drives. MS-DOS 2.11 and NewWord (a word processing program strikingly similar to WordStar) are included. The power pack and internal rechargeable Nicad battery provide more than three hours of continuous operation. There is also a 32K ROM, which includes a number of utilities like a calculator, diary, phone directory, and autodialer. For additional information, contact Morrow at 600 McCormick, San Leandro, California 94577 (415) 430-1970. **Reader Service Number 101.**

Kaypro too has introduced an 11-pound portable, the Kaypro 2000, whose price also begins at $1995. It features full IBM PC compatibility, 256K of RAM, a 25-line screen, a single 3½-inch 720K disk drive (in the minimum configuration), and WordStar and CalcStar bundled in. Its built-in battery pack provides for four hours of continuous operation. The adjustable LCD behaves like an IBM color card with a monochrome adapter. With the addition of an optional base unit (which contains a 5¼-inch drive), the machine can accept standard IBM expansion cards. Kaypro has also announced the 286i model A, a smaller version of its IBM PC/AT compatible 286i, priced at $2995 and including 512K of RAM and GW-BASIC. For further information, contact Kaypro at P.O. Box N, Del Mar, California 92014 (619) 481-4300. **Reader Service Number 103.**

Zenith's new 14.5-pound Z-171 portable PC offers full IBM PC compatibility, 256K of RAM (expandable to 640K), a 25-line flat panel backlit LCD, and two disk drives. The price is $2699. The company has also introduced the Z-200 Advanced PC, a PC/AT compatible model with some extra features, which comes in several configurations priced between $3999 for the single floppy version and $5599 for the 20 MB hard disk version. Other new products include the Z-138, a 24-pound "transportable" with a 7-inch amber display and detachable keyboard, which starts at $2099, and two additions to the Z-150 PC family: the Z-148 and Z-158. Contact Zenith Data Systems at 1000 Milwaukee Avenue, Glenview, Illinois 60025 (312) 391-8949. **Reader Service Number 105.**

You don't have to trade in your aging IBM PC/XT for an AT. Seattle Telecom & Data has announced the PC 286-MTM, a 16-bit iAPX286-based accelerator board with 2.1 MB of bank-switched (paged) memory for the XT. It can support nine consoles and is supported by Xenix, Pick, and Concurrent CP/M. The unit can also support a RAM disk driver, allowing the user to access 1.5 Mb of memory in RAM disk. The price is $1995. Contact the company at 2637 151st Place NE, Redmond, Washington 98052 (206) 883-8440. **Reader Service Number 107.**

Pfaster 286, a 80286-based add-on board that upgrades IBM PCs and PC/XTs to process data faster than an IBM PC/AT without losing the functionality of the native 8088, has been introduced by Phoenix Computer Products. The board is fully PC compatible, runs DOS version 2.0 and higher, and uses the native 8088 as a coprocessor to manage input/output. The price is $2395, with the 80287 available for an additional $350. Contact the vendor at 1420 Providence Highway, Suite 115, Norwood, Massachusetts 02062 (800)344-7200 (in Massachusetts (617) 762-5030). **Reader Service Number 109.**

A dedicated array processor, which plugs into a single expansion slot on the IBM PC, PC/XT, and PC/AT and performs Fast Fourier Transforms 100 to 10,000 times faster than software-computed FFTs, has been introduced by the Ariel Corporation. The price is $1850 (quantity one). Two or more units can be installed in one host and will work in parallel, further increasing throughput. Contact the vendor at 600 West 116th Street, Suite 84, New York, New York 10027 (212) 662-7324. **Reader Service Number 111.**

Hallock Systems, a supplier of coprocessors for Z80 systems, has introduced the Pro68, a Motorola 68000-based coprocessor board for the IBM PC. The board can be configured with between 256K and 3072K of RAM. The CPU runs at 10 MHz with no wait states, and the company says it runs eight times faster than the IBM PC and three times faster than the IBM PC/AT. Both CPM68K and OS9/68000 (a Unix-like operating system) are available, along with an editor, assembler, linker, debugger, and full K&R standard C compiler. Third party software includes all the popular high-level languages. For further information, contact the vendor at 267 North Main Street, Herkimer, New York 13350 (315) 866-7125. **Reader Service Number 113.**

Micro/Sys has introduced the BUS/BRIDGE family, which consists of 14 low-cost development packages that use the IBM PC as the host environment and provide for six target environments, including MULTIBUS (8085 and 8086), VMEbus (68000), STD BUS (Z80 and 8088), and MULTIBUS II (80286). Three software systems are supported: the target processor's assembly language, some of the IBM PC's popular compilers, and a special customized assembler and compiler. The 14 development packages consist of various combinations of the target hardware environments and software systems. Prices run from $1295 to $5595. Contact the vendor at 1011 Grand Central Avenue, Glendale, California 91201. **Reader Service Number 115.**

As everyone knows, APL on the IBM PC requires an 8087 or 80287, which (though they have come down in price somewhat in the last year) are still not exactly cheap. Also, some machines, like the PCjr, don't support the 8087, so until now APL has simply not been available for them. Enter the 8087 Eliminator, a software package from Fort's Software that functionally replaces the math coprocessor but costs only $49 ($75 for the PC/AT version). Contact the vendor at P.O. Box 396, Manhattan, Kansas 66502 (913) 537-2897. **Reader Service Number 117.**

If the PC/AT's 20 Mb disk capacity leaves you unimpressed, perhaps what you need is Reference Technology's CLASIX DataDrive Series 500, a laser disk drive with a capacity of 550 Mb, priced at $1535. The bad news is that every one of those bytes is read only. Just the thing for the Britannica or the unabridged Oxford English Dictionary but not for everyone, despite the very reasonable price. Contact the vendor at 1832 North 55th Street, Boulder, Colorado 80301 (303) 449-4157. **Reader Service Number 121.**

If 550 Mb is more than you need, Optotech has a 400 Mb WORM (Write Once Read Many times) optical disk cartridge available. Contact the vendor at 770 Wooten Road, Suite 109, Colorado Springs, Colorado 80915 (303) 570-7500. **Reader Service Number 123.**

What's a $45 billion company doing in the personal computer market? Whatever it wants to, according to some. What is that company hoping to achieve with TopView? Maybe all those royalty dollars flowing out to Bellevue every quarter are distressing a few souls in Armonk and Boca Ra-

"An important new metaphor from the author of the best book on user interface design."
—Alan Kay,
Apple Fellow, Apple Computer

SNEAK PREVIEW
of a powerful
NEW SOFTWARE METAPHOR
WANTED: People with Imagination, an IBM PC, and $59.95.

by Paul Heckel President, QuickView Systems and author, *Elements of Friendly Software Design*

Rarely does a software product introduce a new conceptual metaphor. VisiCalc introduced the electronic spreadsheet; Thinktank, the electronic outliner; and now *Zoomracks*, the electronic rack. Let me tell you what electronic racks are, why I think they are important, and how you can get to try them risk-free at a savings now and maybe help shape their final form to your liking.

New Metaphor:
Originally designed to keep track of lists, names and addresses, appointments, notes, and other information on portable computers, electronic racks provide a simple, consistent and rich organizational metaphor for data base, text, and other applications.

Zoomracks starts with something familiar: racks—like those filled with time cards next to time clocks in factories. You can see the first line of each card, and take out a card to look at it in detail. You expect the cards in a rack to be in order, several racks to be next to each other; and to be able to move cards from one rack to another.

You might put names and addresses in one, appointments in a second, notes in a third, sales orders in a fourth, memos in a fifth, and archived appointments or notes (moved or copied from the second or third rack) in a sixth rack. To do something with Zoomracks, first ask yourself: "How could I do it with cards in racks?"

ZOOMRACKS SPECIFICATIONS:
- Copy and move fields, cards, and text into different fields, cards and racks.
- Define and change card templates.
- ASCII MS/DOS file format for conversion to other data formats.
- Utilities to convert DBASE II files.
- Macros.
- Simple Wordstar-like editor.
- Easy to learn and easy to use for both occasional and frequent users.
- Display sizes: 6 x 25 to 25 x 80.
- 8 racks on screen, in memory; 30 fields/card; 80 characters/line; 250 lines/field, 20,000 cards/rack.
- Runs on 256K IBM PC.

Windows illuminate like a flashlight in a dark room
Racks are displayed with Smart Zooms. While windows sacrifice the big picture to let you see the detail, Smart Zooms squeeze out the detail to always show you a recognizable big picture—whether a long shot of several racks, a closeup of one rack, or an extreme closeup of a single card.

One time offer for
If you like to stretch new products and influence their final form, we want your feedback. We are making a one time offer of a Sneak Preview Edition of Zoomracks at an affordable price so you can try it and give us your feedback in time to make a difference before we officially introduce Zoomracks in November.

Before developing Zoomracks, Paul Heckel studied what made VisiCalc and other software powerful, useful, easy to use, and successful. He crystallized his thoughts in a book. This is what people are saying about this book, *The Elements of Friendly Software Design:*

"It is the first computing book I've ever read nonstop from cover to cover . . . one of the few books I've read on any topic that actually delivers what it promises . . ."
—Dave Bunnell, PC World Publisher

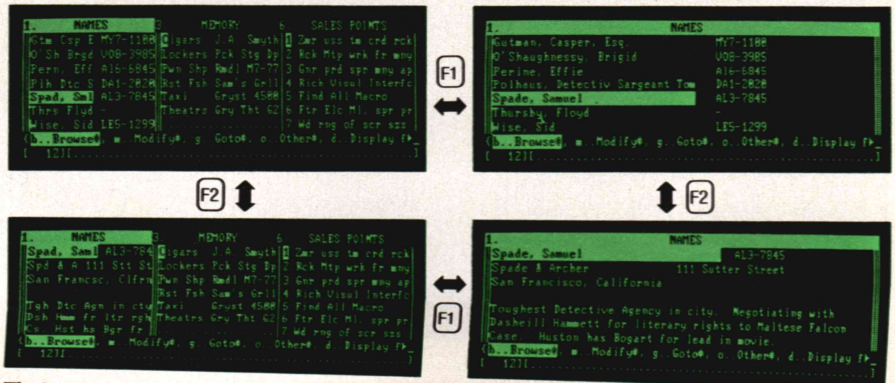
"Entertaining and instructional . . . will affect the way I program from now on."
—David Clark, Byte Magazine

"Informative, useful and entertaining. Will help improve your communications skills in any medium . . ."
—Robert Burton, President, Rolodex

The Elements of Friendly Software Design is available at your local bookstore for $8.95 or by calling 800-443-0100 EXT 341. You can also order by writing QUICKVIEW SYSTEMS, 146 Main St., Suite 404, Los Altos, CA 94022. Add an additional $2.50 for postage and handling. Payment must accompany order.

What you get for $59.95
- The Sneak Preview edition of Zoomracks;
- A free upgrade to the first run edition of Zoomracks;
- An acknowledgement in the user manual *if* you are the *first* to suggest an improvement we use;
- A six-month unconditional moneyback guarantee.

The Wide key (function f1) toggles between displaying the working racks (left two screens) and the current rack full (right two screens). Smart Zooms compress out detail to keep the big picture.



The Yank key (function f2) toggles between displaying the first lines of cards in racks (top two screens), and the current card (bottom two screens). In these pictures Zoomracks is using a 10 by 60 screen.

To order the Sneak Preview Edition
$59.95 copy protected
$79.95 copy unprotected

CALL
800-443-0100 EXT 341
OR
WRITE
QUICKVIEW SYSTEMS
146 Main Street,
Suite 404
Los Altos, CA 94022

ton, or maybe something more sinister is afoot. Who knows? But software developers are starting to bring out products with TopView in mind—although at this point TopView is mostly unfulfilled promise. Lattice has introduced the Lattice TopView Toolbasket for programmers writing applications to take advantage of TopView's multitasking window environment. There are 70 C functions to control window, cursor, and pointer operations, as well as printer control, cut and paste, and debugging functions. Memory requirements are 256K, but 512K are recommended. The price is $250 and the source code is available for $250 more. For further information, contact Lattice at P.O. Box 3072, Glen Ellyn, Ilinois 60138 (312) 858-7950. **Reader Service Number 125.**

Matrix Software Technology has introduced Synergy, a 12K operating system with Macintosh-like desktop and windows, which runs under either DOS or TopView and allows up to six programs to run concurrently in windows without modification. Synergy can also run alone, and because it requires only 12K of RAM, it will run on any PC and outperform TopView, according to the vendor, who can be contacted at 50 Milk Street, Boston, Massachusetts 02109. **Reader Service Number 127.**

Taking another approach, Quarterdeck Office Systems has introduced DESQVIEW, a new version of its floppy-based DESQ memory-resident multitasking software integrator. Priced at $99.95, it supports both keyboard and mouse. Contact the vendor at 1918 Main Street, Santa Monica, California 90405 (213) 392-9851. **Reader Service Number 129.**

The New York Amateur Computer Club has published a catalog of its PC/Blue Library: over 560 public domain programs for the IBM PC and compatibles. The catalog is priced at $5 ($7 for overseas airmail) including postage. The diskettes are $7 postpaid (North America) and $10 overseas air. Membership dues are $15 per year. Contact the club at P.O. Box 106, Church Street Station, New York, New York 10008. **Reader Service Number 133.**

Flagstaff Engineering is now distributing the DISKETTE CONNECTION and the WORD CONNECTION. The DISKETTE CONNECTION is an 8-inch stand-alone drive system with controller that allows a mini or mainframe to read from or write to a PC unit. The WORD CONNECTION is a set of software programs that provide the ability to read and write text documents from most word processing systems using an IBM PC equipped with the DISKETTE CONNECTION. All of the WORD CONNECTION programs transfer documents to a PCDOS file using IBM's Revisable Form Text standards for document architecture (DCA). For more information write Flagstaff Engineering, Box 1970 Flagstaff, AZ 86002, (602) 774-5188. **Reader Service Number 135.**

Copy-protected software is a royal pain in the ASCII, but with the help of TranSec Systems' UNlock, you can banish the pain forever, according to the vendor. The program "removes copy protection" and provides standard nonprotected DOS copies, which enable the user conveniently to run the software from a hard disk or RAM disk. The original disk need no longer be present in drive A when the program is run. The price is $49.95. For further information, contact the vendor at 701 East Plantation Circle, Plantation, Florida 33324 (305) 474-7548. **Reader Service Number 137.**

KDS Corporation has announced its Knowledge Delivery System, an artificial intelligence program that produces expert systems of up to 16,000 production rules and 256,000 facts from up to 4,096 case histories. The fully menu-driven system communicates with the user in English. The cost is $495–$795, depending on options. Contact KDS at 934 Hunter Road, Wilmette, Illinois 60091 (312) 251-2621. **Reader Service Number 139.**

CALL ADVENT

SERVICES

File Transfer Service: Advent provides a service beyond the ability of any format conversion software! We can transfer files between MS-DOS/PC-DOS, CP/M and other operating systems in 300 different 3 1/2", 5 1/4" and 8" formats. Includes Apple and Mac, Apricot, Data General One, Kaypro 2000, Eagle, Epson QX-10 & PX-8 (ROM), HP-150, and North Star computers.
.... **Call for information and pricing**

ENGINEERING SOFTWARE

ACNAP: A stand-alone Electronics Circuit Analysis Program for use with passive and active circuits consisting of resistors, capacitors, inductors, transistors, op-amps, FETs, etc. Features menu driven and very fast processing times with circuits saved to disk for later use or editing.
ACNAP (CP/M & MS-DOS) **$69.95**

DCNAP: Stand-alone DC circuit analysis program for use with passive and active circuits containing resistors, voltage sources, independent and dependent current sources. Fast, menu-driven program with circuit saved to disk for later use or editing.
DCNAP (CP/M & MS-DOS) **$69.95**

Plotpro: Scientific graph printing program. Prints on 80 or 132 column printer. Create linear, semi-logarithmic, and full logarithmic plots with one or two Y axes in auto or forced scale.
Plotpro (CP/M & MS-DOS) **$69.95**

SPP: This Signal Processing Program contains an intergrated set of routines which analyze linear and non-linear systems and circuits and their effects on user specified time domain waveforms. Based on a 512 point Fast Fourier Transform and its inverse. Linear processing is in frequency domain and non-linear processing is in time domain.
SPP (CP/M & MS-DOS) **$69.95**

HOME & BUSINESS

Checks & Balances version 3.6: A complete personal checking program or business register. 3.6 is completely rewritten and includes check writer and data base. It is easy to learn and use with new features including more types of reports. Complete with new 180 page manual. An excellent money manager!
Checks & Balances (CP/M & MS-DOS) **$74.95**

Basic Business: Stand aside DAC! Here is the ultimate price / performance package for small to medium sized businesses. Not a chopped down system, but a full featured accounting system which sold for over $700. Includes GL, AR, AP, Payroll, Inventory Control, Order Entry, Purchase Order Journal. Data files are dBase II compatible and are transportable between the CP/M and MS-DOS versions.
Basic Business (CP/M & MS-DOS) **$89.95**

SOFTWARE UTILITIES

Autodiff: File difference detector. This program finds insertions, deletions, and changes between any two files. Autodiff can mark the file, display, or print the differences, and more!
Autodiff (CP/M) **$29.95**

CP/M DateStamper: Automatically stamp your files with the date it is created, last read, or modified. Works without a Real Time Clock, or with many clocks currently on the market. Utilities are included to allow copying, erasing, or renaming files based on time and date. A time logging utility is included to record computer usage for business/tax purposes.
DateStamper (CP/M) **$49.95**

Pack and Crypt: Two program set. Pack compresses and expands files on disk to save space. Crypt encodes files to provide security for sensitive data. Both are ideal for use with modem transfers.
Pack and Crypt (CP/M & MS-DOS) **$24.95**

Sidekick: One of the most popular programs ever written. Use Sidekick as a calculator, notepad, appointment calendar, auto dialer, ASCII conversion table and much more. On-line help if you forget any of Sidekick's many functions.
Sidekick (MS-DOS) **$54.95**

SmartKey: Saves you time. Makes every software program you use easier. Can reduce keystrokes by more than 50% by redefining any key to be any combination of characters or commands that you desire.
SmartKey (CP/M & MS-DOS) **$49.95**

SmartPrint: A powerful add-on to SmartKey, SmartPrint is a versatile writing tool designed to give you full access to your printer's features such as wide, bold, condensed, underlined, subscript, superscript, and more. Works great with programs like WordStar.
SmartPrint (CP/M & MS-DOS) .... **FREE WITH SMARTKEY PURCHASE**

Uniform: Your Computer can read and write up to 80 CP/M and MS-DOS / PC-DOS disk formats. Versions available for most popular CP/M and MS-DOS computers. Specify your host computer when ordering.
Uniform (CP/M & MS-DOS) **$69.95**

XTREE: Directory maintenance program that graphically displays subdirectories and file-name paths. Complete control of your directory including delete, rename, view, list or show. A must for your IBM or compatible.
XTREE (MS-DOS) **$49.95**

Circle no. **4** on reader service card.

FX - Epson Corp. CP/M - DRI, MS-DOS - MicroSoft, PC-DOS - IBM Corp., dBASE II - Ashton Tate.

Super Zap: Disk patch and dump program. If you have used DU, you will love this menu driven marvel!
Super Zap (CP/M) **$24.95**

ZP/EM: Run almost any CP/M program on your IBM or clone. Use with Media Master or Uniform to allow programs on CP/M disk formats to run directly on your IBM or compatible computer.
ZP/EM (MS-DOS) **$59.95**

PROGRAMMING LANGUAGES

C/80 Ver 3.1: Full featured C compiler and runtime library. One of the fastest on the market. Mathpak is included for true 32 bit floating point and signed integers.
C/80 Ver. 3.1 (CP/M) **$79.90**

Toolworks C: This compiler is a complete subset of C. The two-pass compiler produces relocatable object files (.obj) which are compatible with the MS-DOS LINK program. Mathpak is included for true 32 bit floating point and signed intergers.
Toolworks C Compiler (MS-DOS) **$79.90**

Turbo Pascal: Borland version 3.0. The best Pascal compiler on the market.
Turbo Pascal (CP/M & MS-DOS) **$69.95**

Turbo Toolbox: Set of 3 utilities for use with Turbo Pascal.
Turbo Toolbox (CP/M & MS-DOS) **$54.95**

Turbo Tutor: Teaches step-by-step how to use Turbo Pascal.
Turbo Tutor (CP/M & MS-DOS) **$34.95**

Turbo Graphics: Provides full graphics management for producing windows, pie and pie charts, circles and other geometric shapes with Turbo Pascal.
Turbo Graphics (MS-DOS) **$54.95**

TEXT EDITING

Punctuation & Style: Improves your writing by catching unbalanced quotes, parentheses and brackets, improper abbreviations, capitalization, sentence structure, much more. It's like having your own copy editor!
Punctuation & Style (CP/M & MS-DOS) **$125.00**

Word Finder: This powerful 90,000 word Thesaurus allows you to select the best word for the application. Works inside WordStar for greater ease of use. Instantly searches its dictionary, then displays synonyms, and automatically deletes the "wrong" word and replaces it with the "right" word. Requires 380K disk storage.
Word Finder (CP/M) **$79.95**

Wordpatch: Print files with tiny, compressed, wide, or wide compressed type faces, 5 sizes of italic, real superscripts and subscripts, and 6, 7, and 8 lines per inch spacing. No new print controls to learn. Supports most popular dot matrix printers. A must for WordStar users!
Wordpatch (CP/M & MS-DOS) **$49.95**

The Word Plus: The ultimate spelling checker. Not only finds misspelled words but shows you correct spelling options, shows the word in context, allows you to build dictionaries of special words you use, and much more.
The Word Plus (MS-DOS) **$150.00**

HARDWARE & SUPPLIES

Finger Print "Letter Writer": Unleash your Epson FX series printer. Add near-letter-quality print, IBM and/or Apple Graphics printer emulation, plus 16 other print functions! Three replacement chips quickly fit inside Epson FX series printers. Easy installation. Does not void printer warranty.
Finger Print "LetterWriter" **$79.95**

Diskettes, Double Density:

Maxell box of 10 with storage box.		3M box of 10	
Single Sided	**$19.95**	**Single Sided**	**$22.95**
Double Sided	**$23.95**	**Double Sided**	**$26.95**

Economy Diskettes: package of 25 including tyvek sleeves.
Single Sided **$29.50**
Double Sided **$31.25**

Call or write for our FREE catalog

All items are warranteed for 90 days. 30 day money back guarantee if not completely satisfied. Guarantee for software applies only if diskette seal is intact. Visa and MasterCard are welcome. Please add 2.00 freight per total order and 2.00 for COD orders. California residents please add 6% sales tax. Prices, availability and specifications subject to change without notice.

CALL TODAY **National (800) 821-8778**
California (800) 521-7182

Hours: Mon - Fri 8 am - 5 pm PDT
DEALER INQUIRIES WELCOME.

advent products inc.
3154-F E. La Palma Ave.
Anaheim, CA 92806
(714) 630-0446

CP/M

For some time, *DDJ* readers have been writing to express concern about Digital Research's apparently waning level of support for CP/M 80. Colonial Data Systems is now making available CP/M 2.2 and CP/M Plus in the same unconfigured packages that DRI supplied directly in the past. The prices are $75 for CP/M 2.2 and $275 for CP/M Plus. Contact the vendor at 80 Picket District Road, New Milford, Connecticut 06776 (203) 355-3178. **Reader Service Number 163.**

A hand-sized computer from Britain with CP/M compatibility and up to 256K of RAM has been announced by AMS Numerics. The LCD screen can display two or four lines of 16 characters, and the keyboard is available with either 20 or 42 keys. Programs and data can be downloaded to the unit via its built-in RS232 port at rates up to 9600 baud. A parallel port allows interfacing with other electronic units, and there is also a separate bar code reader port. The unit weighs 29 ounces and operates in temperatures from 0 – 40°C. Power is supplied by a rechargeable, removeable battery pack, which can power the unit for up to three months at a time. Contact the vendor at Wallstreams Lane, Worsthorne Village, near Burnley in Lancashire, BB10 3PP, England. **Reader Service Number 165.**

Spectravideo has introduced an 11-pound, battery rechargeable, CP/M 2.2-based lap-sized portable computer with a 25-line display, a built-in 3.5-inch drive (360K formatted), and six pieces of bundled software (WordStar, ReportStar, CalcStar, MailMerge, DataStar, and Scheduler Plus) for a list price of under $1000. The product is scheduled for delivery in September 1985. The company also will offer three "transportable" CP/M computers and two MSDOS desktop machines. Contact the vendor at 3300 Seldon Court #10, Fremont, California 94539 (415) 490-4300. **Reader Service Number 167.**

The Private Line is a software-based DES system that can process any CP/M file at the rate of 10K per minute with a 4 MHz cpu. It will also overwrite a file with binary zeros and purge it from the directory, as well as display a file in hex format. Contact the vendor, Everett Enterprises, at P.O. Box 193, Bath, NC 27808 (919) 923-5621. **Reader Service Number 169.**

Note

In the July "Of Interest" column we stated that the EL286-88 Processor Converter from Edsun Labs allows you to replace directly an 8088 microprocessor with an 80286. A more accurate description would be to say that the EL286-88 is a VLSI component that greatly simplifies the hardware design of an 80286 upgrade board by converting 8-bit 8088 bus signals to those of the 16-bit 80286 while accomodating asynchronous clock rates.

DDJ

ADVERTISER INDEX

* This advertiser prefers to be contacted directly: see ad for phone number.

Advertising Sales Offices

East Coast
Walter Andrzejewski (617) 567-8361

Midwest/West Central
Michele Beaty (317) 875-0557

Northern California/Northwest
Lisa Boudreau (415) 424-0600

Southern California/Southwest
Beth Dudas (714) 643-9439

Advertising Director
Shawn Horst (415) 424-0600

PC WEEK'S PRODUCT OF THE YEAR
PC MAGAZINE'S AWARD FOR TECHNICAL EXCELLENCE

Borland Introduces the Laws of *TURBO DYNAMICS*™

Laws That Work Like Magic. Whether considering technological excellence, or innovation in areas such as pricing, not copy-protection, licensing agreements, site licenses, 60 day money-back guarantee—Borland is clearly recognized as the software industry leader. The following three laws of "*Turbo Dynamics*"™ exemplify our pledge for excellence.

1ST LAW:

SPEED, POWER AND PRICE.

Borland products are known to be fast, powerful and to deliver an incredible price performance ratio. We only believe in absolutely superb software at rock bottom prices.

2ND LAW:

NOT COPY-PROTECTED SOFTWARE AND REASONABLE LICENSING AGREEMENTS.

We will always offer not copy-protected versions of our software. Also, our licensing agreement is now so simple that even a child can understand it.

3RD LAW:

60 DAY MONEY-BACK GUARANTEE.

This third law is actually a first in the industry! We are so sure that you will love our software that all of our products now come backed with a 60 day money-back guarantee. No questions asked.

Turbo Dynamics Applies to Turbo Pascal. Borland's Pascal family of products is growing by leaps and bounds. You can now join hundreds of thousands of users and enter the world of Turbo Pascal programming. And remember, all three laws of *Turbo Dynamics* apply to all Borland products.

TURBO PASCAL™ $69.95

The industry standard. With more than 350,000 users worldwide Turbo Pascal is the industry's de facto standard. Turbo Pascal is praised by more engineers, hobbyists, students and professional programmers than any other development environment in the history of microcomputing. And yet, Turbo Pascal is simple and fun to use. **Free spreadsheet** included on every Turbo disk with ready-to-compile source code. ***Options:*** We offer the exciting Binary Coded Decimal (BCD) option for your business applications as well as an 8087 option for your number-crunching applications at a very low charge. Please refer to the coupon. ***Portability.*** Turbo Pascal is available today for most computers running PC-DOS, MS-DOS, CP/M-80 or CP/M-86. ***Jeff Duntemann, PC Magazine:*** "In its simplicity it achieves an elegance that no other language compiler has ever displayed."

TURBO GRAPHIX TOOLBOX™ $54.95

High resolution monochrome graphics for the IBM PC.
The Turbo Graphix Toolbox will give even a beginning programmer the expert's edge. It's a complete library of Pascal procedures and functions. Tools that will allow you to draw and hatch pie charts, bar charts, circles, rectangles and a full range of geometric shapes. Procedures that will save and restore graphic images to and from disk. And much, much, more. You may incorporate part or all of these tools in your programs and yet we won't charge you any royalties. Best of all, these functions and procedures come complete with commented source code on disk ready to compile.

TURBO TUTOR™ $34.95

From start to finish in 300 pages. Turbo Tutor is for everyone from novice to expert. Even if you've never programmed before Turbo Tutor will get you started right away. ***A must.*** You'll find the source code for all the examples in the book on the accompanying disk ready to compile. Turbo Tutor might be the only reference on Pascal and programming you'll ever need.

TURBO DATABASE TOOLBOX™ $54.95

The Turbo Database Toolbox is the perfect complement to Turbo Pascal. It contains a complete library of Pascal procedures that allows you to sort and search your data and build powerful applications. It's another Borland set of tools that will give the beginning programmer the expert's edge. ***Get started right away: free database!*** Included on every Toolbox disk is the source code to a working data base which demonstrates how powerful and easy to use our search system, Turbo-Access, really is. Modify it to suit your individual needs or just compile it and run. ***Remember, no royalties!***

NOT COPY-PROTECTED

TURBO PASCAL FAMILY

Available at better dealers nationwide. Call (800) 556-2283 for the dealer nearest you. To order by Credit Card call (800) 255-8008, CA (800) 742-1133

60 DAY MONEY-BACK GUARANTEE

Carefully Describe your Computer System!

Mine is: ___ 8 bit ___ 16 bit
I Use: ___ PC-DOS ___ MS-DOS ___ CP/M 80 ___ CP/M 86

My computers' name/model is: ____________

The disk size I use is:
☐ 3½" ☐ 5¼" ☐ 8"

Name: ____________
Shipping Address: ____________
____
City: ____________
State: ________ Zip: ________
Telephone: ____________

Pascal 3.0	$ 69.95	____
Pascal w/8087	$109.90	____
Pascal w/BCD	$109.90	____
Pascal w/8087 & BCD	$124.95	____
Turbo Database Toolbox	$ 54.95	____
Turbo Graphix	$ 54.95	____
Turbo Tutor	$ 34.95	____

*These prices include shipping to all U.S. cities. All foreign orders add $10 per product ordered.

Amount: (CA 6% tax) ____________
Payment: VISA MC BankDraft Check
Credit Card Expir. Date: ______ / ______
Card #: ____________

11

COD's and Purchase Orders WILL NOT be accepted by Borland. California residents: add 6% sales tax. Outside USA: add $10 and make payment by bank draft, payable in US dollars drawn on a US bank.

BORLAND INTERNATIONAL

4585 Scotts Valley Drive, Scotts Valley CA 95066
Phone (408) 438-8400 Telex 172373

Circle **no. 14** on reader service card.